Technical Report 1139

Using Special-purpose Computing to Examine Chaotic Behavior in Nonlinear Mappings

Jason Nieh

MIT Artificial Intelligence Laboratory

4

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AI-TR 1139
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Using Special-purpose Computing to Examine Chaotic Behavior in Nonlinear Mappings

5. TYPE OF REPORT & PERIOD COVERED: technical report

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Jason Nieh

8. CONTRACT OR GRANT NUMBER(s): N00014-86-K-0180

9. PERFORMING ORGANIZATION NAME AND ADDRESS:
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:

11. CONTROLLING OFFICE NAME AND ADDRESS:
Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

12. REPORT DATE: September 1989

13. NUMBER OF PAGES: 72

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
Office of Naval Research
Information Systems
Arlington, VA 22217

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

chaos
nonlinear mappings
numerical computation
computer architecture
standard map
standard map machine

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See Reverse

Block 20 cont'd

Most physical phenomena are nonlinear in nature and exhibit the complicated and seemingly random behavior known as chaos. Studying chaotic behavior in nonlinear systems requires numerous computations in order to simulate the behavior of such systems. The Standard Map Machine (SMM) was designed and implemented as a special computer for performing these intensive computations with high-speed and high-precision. SMM's impressive performance is due to its simple architecture specialized to the numerical computations required of nonlinear systems. This report discusses the design and implementation of the Standard Map Machine and its use in the study of nonlinear mappings, in particular, the study of the standard map.

Using Special-purpose Computing to Examine Chaotic Behavior in Nonlinear Mappings

by

Jason Nieh

Artificial Intelligence Laboratory
and
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Originally S.B. thesis, M.I.T. Department of Electrical Engineering and Computer Science, May, 1989. Revised September, 1989.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

Using Special-purpose Computing to Examine Chaotic Behavior in Nonlinear Mappings

by
Jason Nieh

Artificial Intelligence Laboratory
and
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Originally S.B. thesis, M.I.T. Department of Electrical Engineering and Computer Science, May, 1989. Revised September, 1989.

ABSTRACT

Most physical phenomena are nonlinear in nature and exhibit the complicated and seemingly random behavior known as chaos. Studying chaotic behavior in nonlinear systems requires numerous computations in order to simulate the behavior of such systems. The Standard Map Machine (SMM) was designed and implemented as a special computer for performing these intensive computations with high-speed and high-precision. SMM's impressive performance is due to its simple architecture specialized to the numerical computations required of nonlinear systems. This report discusses the design and implementation of the Standard Map Machine and its use in the study of nonlinear mappings, in particular, the study of the standard map.

Keywords:
chaos, nonlinear mappings, numerical computations,
computer architecture, standard map, Standard Map Machine

Acknowledgements

I am grateful to my thesis supervisor, Professor Gerald J. Sussman. He first suggested to me the idea of special-purpose computing for dynamics and provided guidance in considering the applications for the special computer and the methods of computation. Many thanks for all of his advice and encouragement.

Much gratitude is due to Andrew Berlin for his suggestions and encouragement. He suggested a number of useful considerations in the initial construction of the machine and also helped with the debugging of the hardware, pointing out possible analog problems due to the speed of the machine. Thanks also for his help in debugging the program used to calcuate Lyapunov exponents of the standard map, and for reading initial drafts of this report.

I thank Brian LaMacchia, who wrote much of the software support for the host interface as well as the microcode assembler for SMM. He also patiently helped with the debugging of the system, and gave useful instruction in formating this report; several of the figures and tables shown in here were made by him.

Special thanks to Henry Wu, who helped with the initial design of SMM and provided valuable suggestions and assistance in implementing the machine. He educated me on many of the design considerations necessary in high-speed digital design and provided numerous insightful suggestions about the implementation, especially during the debugging of the hardware, which was made easier because of software tools that he created.

Thanks also to Panayotis Skordos and Gerald Roylance for suggestions on numerical methods, Mike Eisenberg and Professor Jack Wisdom for explanations about nonlinear phenomena, William McAllister for suggestions about hardware debugging, Bill Rozas for helping with the final stages of debugging the computer system, and Professor Harold Abelson for helpful suggestions in writing this report.

Finally, I would like to thank Mom and Dad for the opportunity to learn, Uncle for advice during these college years, and Him, who makes all things possible and gives life meaning.

Contents

List of Figures

Chapter 1

Introduction

There exists in nature a wide variety of phenomena that exhibit complicated and seemingly random behavior. Examples abound all around us - just take a moment to gaze at the cloud patterns in the sky or swirl the cream in your cup of coffee. Useful theoretical descriptions of these complex phenomena have in the past proven difficult to construct due to the nonlinearity of the mathematical equations which model these physical systems. But in recent years, much progress has been made in this area of dynamics with the use of high-speed digital computers. These powerful tools enable us to identify and explore ordered patterns in the behavior of systems by combining numerical experiments with mathematical analysis. We have found that many physical systems exhibit an incredible spectrum of dynamical behavior ranging from the ordered to the unpredictable. The technical term that is used to describe this irregular, unpredictable behavior of deterministic systems is *chaos*. It may not be surprising that dynamical systems with many degrees of freedom can behave in an unpredictable fashion for all practical purposes, but what is exciting is that chaos manifests itself in even systems with few degrees of freedom; the observed behavior appears just as complicated as the many degrees of freedom cases.

For a digital computer to simulate such systems, the behavior must be expressed in terms of a discrete process that allows the computer to perform the computation in distinct steps. Such a discrete process is essentially an iterated mapping, representable by a set of difference equations. In other

words, discrete mappings as well as continuous flows are represented as mappings from a computational point of view. For this reason, as well as the reason that mappings are interesting systems in and of themselves, it is important to gain a better understanding of these discrete systems in the study of chaotic behavior. One mapping of particular interest is the standard map. Although just a two-dimensional set of difference equations, it can exhibit complex chaotic behavior and can be used to represent a host of real physical systems.

In order to examine the behavior of such mappings, it is often necessary to perform large numbers of iterations of the equations of the system. Because of these computational requirements, I have designed and implemented the Standard Map Machine, a special computer to perform the mappings with high-speed and high-precision. It was designed as a backend processor that could be solely dedicated to the iteration of nonlinear mappings to study chaotic behavior. The prototype implementation performs at an average rate of about 2.5 MFlops for the class of problems that it was designed to solve, dissipating approximately 40 Watts of power. This machine was used to examine the behavior of the standard map by computing the Lyapunov exponents for various initial conditions. These results and the novel computer design and implementation are presented in the following sections.

Chapter 2

Nonlinear Mappings: The Standard Map

A mapping is a discrete representation of a dynamical system, consisting of a set of difference equations. A nonlinear mapping is a mapping that consists of a set of difference equations that contain nonlinear terms. In particular, we would like to consider nonlinear mappings of few dimensions. The study of such systems is of great interest because of several reasons. Simple nonlinear mappings represent numerous simple physical systems that are all around us, and as yet are still not well understood. A variety of techniques have been developed for studying few dimensional systems that help shed light on the nature of chaos in such systems. Furthermore, the study of simple nonlinear mappings brings us insight about the complicated behavior of many dimensional systems.

One example of such a mapping is the *standard map*. The standard map is a set of two difference equations:

$$x_{n+1} = (x_n + y_{n+1}) \; mod \; 2\pi \tag{2.1}$$

$$y_{n+1} = (y_n + k \sin(x_n)) \; mod \; 2\pi \tag{2.2}$$

The nonlinearity of the system is introduced by the $\sin(x_n)$ term, and can be varied by changing the value of the parameter k. Note that the calculation of the state variables modulo 2π exploits the natural periodicity of the system.

This can be seen by the fact that incrementing the current state variables by 2π changes the next state variables by precisely the same amount.

Perhaps the name "standard map" is derived from the fact that this simple mapping can be used to represent a large number of physical systems. It provides an approximate description to the general class of one-dimensional nonlinear oscillators subject to periodic perturbations. One such physical system that is described by this set of equations is a rigid rotor periodically perturbed by a person kicking it [6]. The rigid rotor can be represented by the standard map if we let x_n represent the angular position of the rotor at the time of the nth kick, y_n represent the angular velocity of the rotor just before the nth kick, and $k\sin(x_n)$ represent the strength and direction of the kicks.

In investigating the behavior of nonlinear mappings, it is useful to observe the evolution of the state of the system in phase-space, given a set of initial conditions. For the case of the standard map, we would like to observe the trajectory of the phase-point (x_n, y_n), given an initial state (x_0, y_0) and a value for the parameter k. The phase-space portraits of the standard map for the initial conditions $x_0 = .05$, $y_0 = .05$, and various values of k are shown in Figures 2.1 through 2.4.

As the coefficient k of the nonlinear term increases, the resulting phase-space portrait exhibits more and more complex behavior. For $k = 0$, the phase-space portrait is a straight line. For small values of k, in this case $k = .2$, the phase-space portrait is a curve. In both of these cases, the trajectory of the phase-point is regular and predictable. The system is stable for these initial conditions and the small value of k. For larger values of k, in this case $k = 1$, the phase-space portrait exhibits beautiful patterns of regions of order and regions of disorder. The dark regions of disorder are chaotic. For very large k, the portrait seems to be entirely random. Note that for larger values of k, the system exhibits large regions of instability for these initial conditions.

There is much evidence that indicates the long-term behavior of chaotic systems cannot be simply determined by just studying short-term behavior [10]. Given the rich behavior that can arise in these systems, we would like to study the evolution of different phase-points for long durations. For instance, one intriguing problem involves the standard map with a varying k value. Suppose we start with initial conditions x_0, y_0, k_0 such that this orbit is regular,

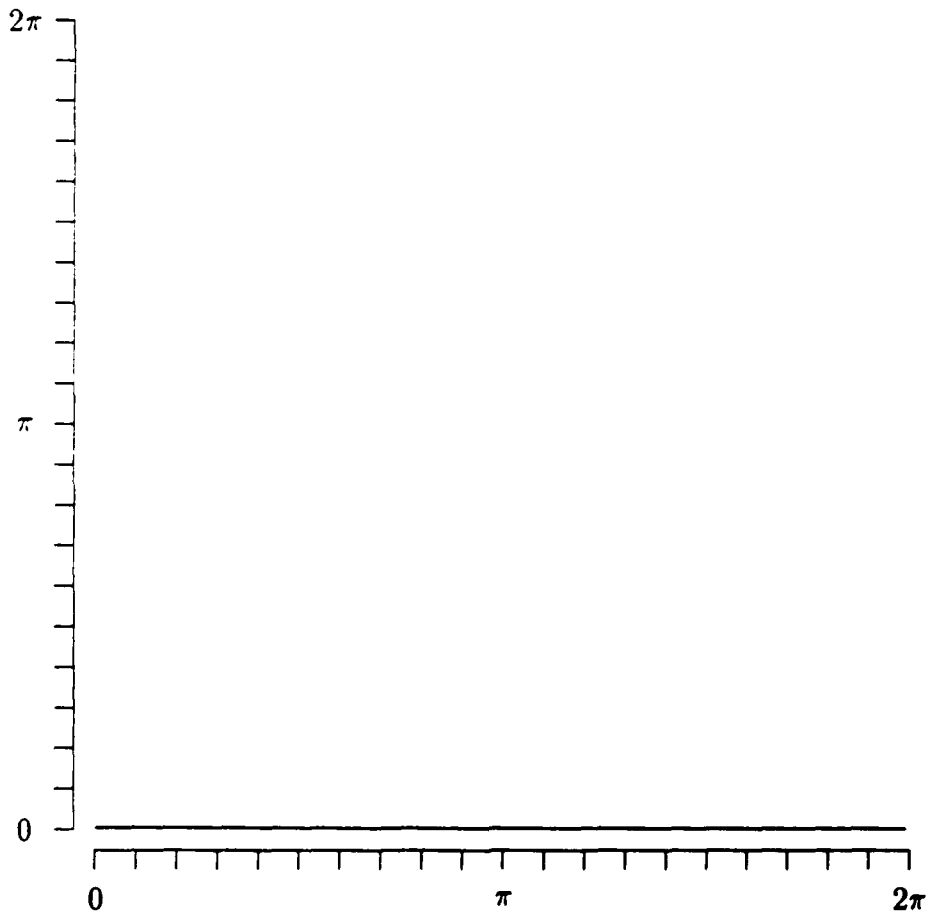

Figure 2.1: Phase-space Portrait for k = 0

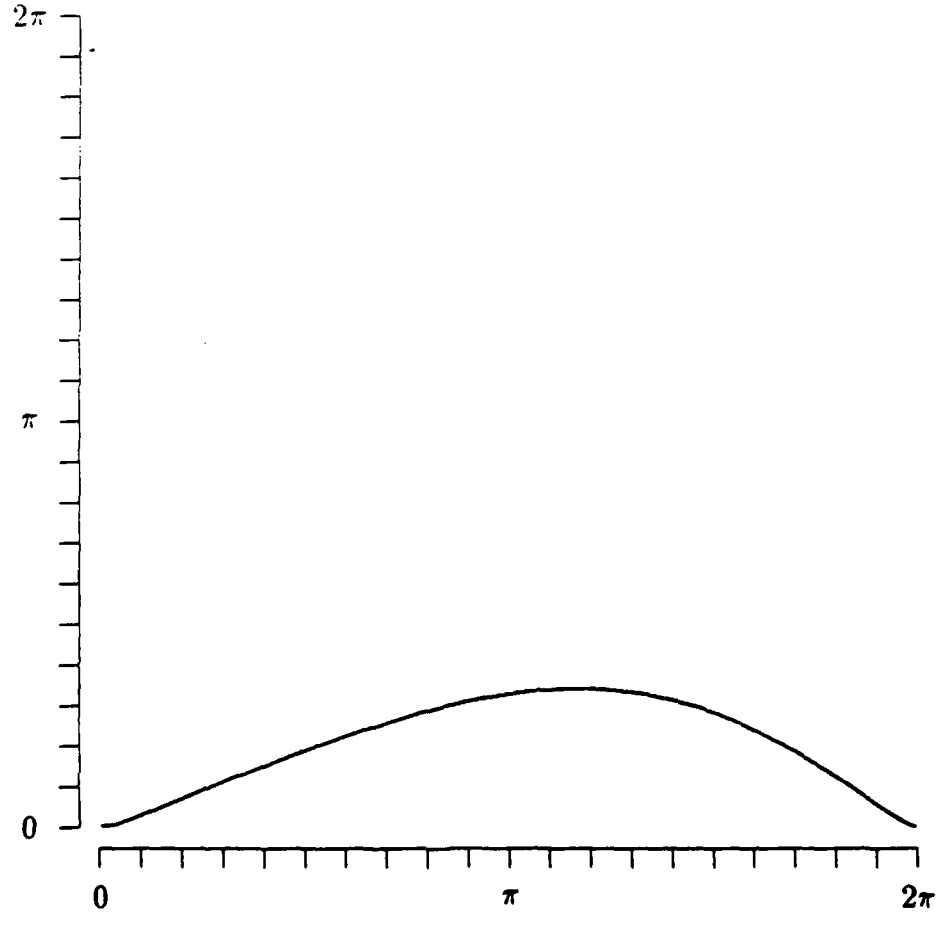

Figure 2.2: Phase-space Portrait k = .2

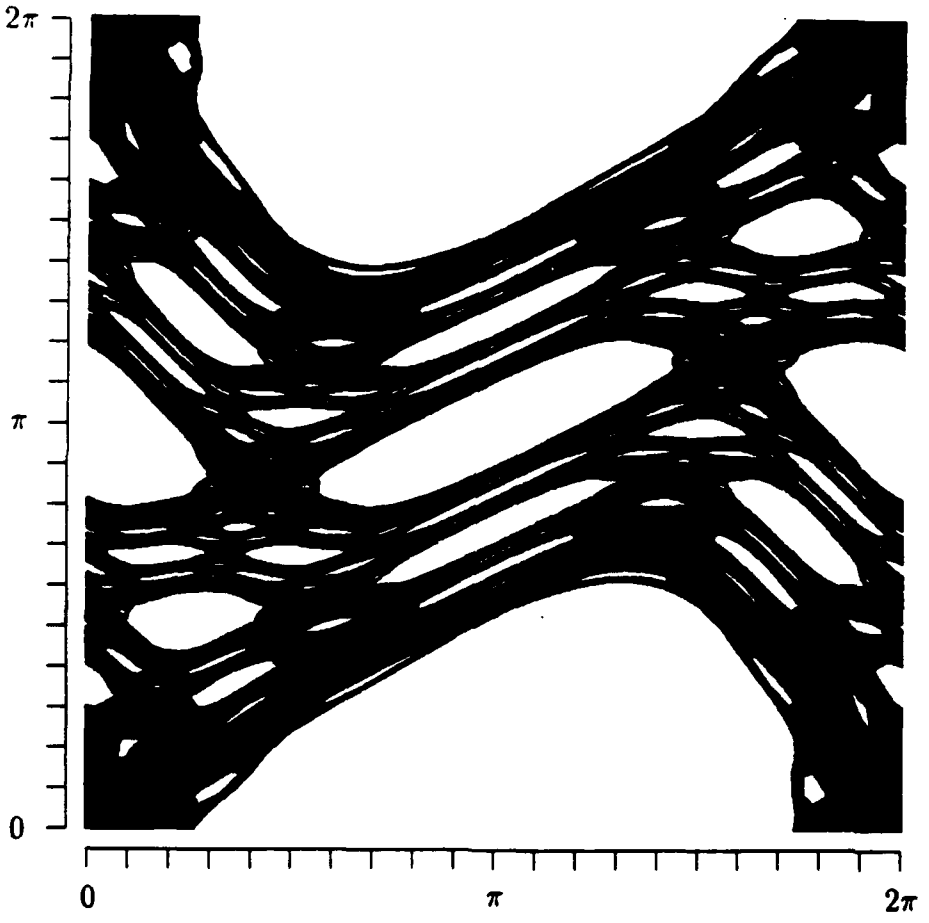

Figure 2.3: Phase-space Portrait k = 1

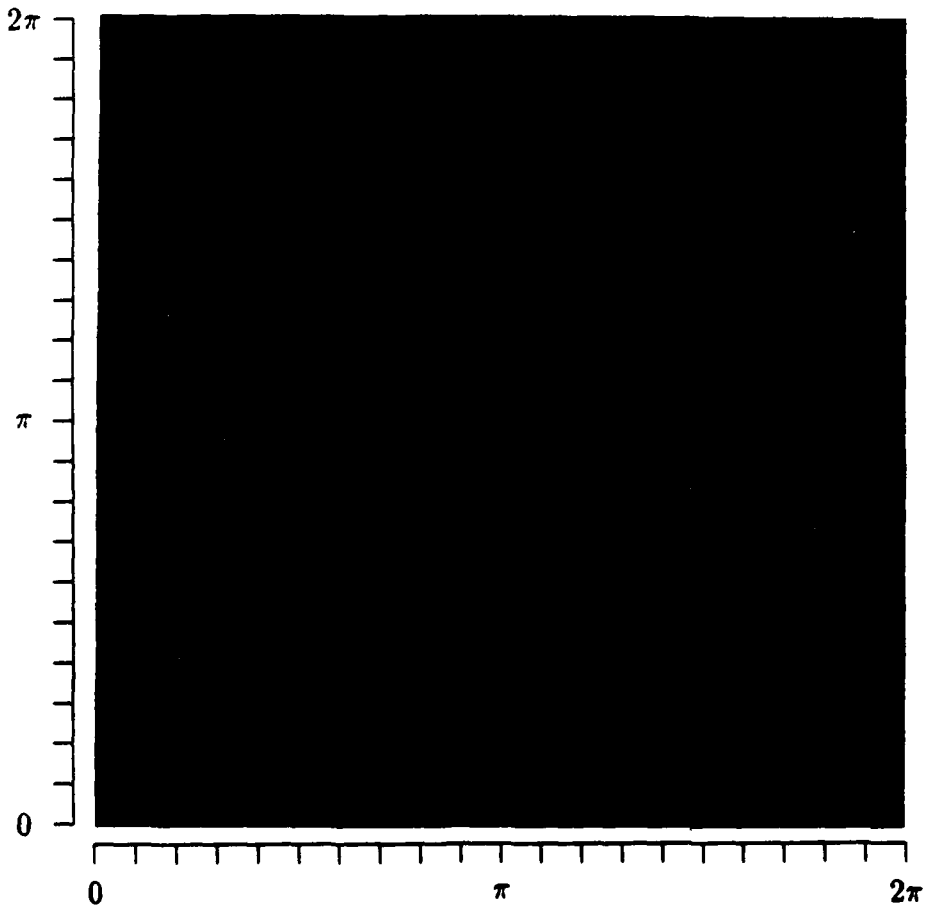

Figure 2.4: Phase-space Portrait for k = 10

but for initial conditions x_0, y_0, k_1, we get a chaotic orbit. If we slowly vary k from k_0 to k_1 and back to k_0 while iterating the map, is the final orbit for $k = k_0$ regular or chaotic? The answer to this question is not known in general.

However, such investigations require large numbers of iterations of the mappings. Fast computing facilities would be necessary, but most conventional machines are too slow and supercomputing power is costly. Due to these intensive computational requirements, a special computer was designed to perform the computation of nonlinear mappings with high-speed and high-precision. This special computer is the Standard Map Machine.

Chapter 3

A Special-purpose Computer: The Standard Map Machine

3.1 Initial Considerations

The Standard Map Machine (SMM) was designed as a solution to the computational problem of studying the long-term behavior of simple nonlinear systems. A number of initial considerations were taken into account in designing this machine. These included size, precision, availability of parts, and kinds of operations needed to be performed. In terms of size, it was desired that the machine would be relatively small and simple, something that could be used as a backend processor to a host computer, such as a conventional workstation, and something that could be implemented in a relatively short amount of time. With regard to precision of numerical calculations, it would be necessary to be able to compute with 64-bit double precision accuracy in order to minimize roundoff errors that could potentially be fatal for large numbers of iterations of a mapping. Concerning the availability of parts, the main computational unit was chosen to be a set of Weitek floating point chips, consisting of an ALU and a multiplier. In terms of the computations that needed to be performed, the machine needed to be able to perform standard functions that were common in simple nonlinear mappings.

To determine the necessary functionality of the machine, we consider the

class of nonlinear systems that SMM would need to compute: simple nonlinear mappings. A good example to use would be the standard map, described in the previous chapter. It was evident from observing this pair of difference equations that addition, multiplication, modulo n, and elementary trigonometric functions needed to be performed. Addition, multiplication, and modulo n could all be performed as straightforward standard operations using the floating point arithmetic units. In order to compute trigonometric functions, a Taylor series approximation with Chebyshev economization was used to perform the calculation with the fewest number of terms for the necessary precision [9]. Thus, the sine function in the standard map can be calculated using the following series expansion:

$$\begin{aligned} \sin(x) &= x(s_1 + x^2(s_3 + x^2(s_5 + \\ &\quad x^2(s_7 + x^2(s_9 + x^2(s_{11} + s_{13}x^2)))))) \end{aligned} \tag{3.1}$$

The numerical coefficients are given in Appendix C. It was determined that for 64-bit IEEE double precision accuracy, the sine series approximation required seven coefficients. Note that the sine polynomial expansion is expressed such that the number of multiplications is minimized, thus minimizing the latency for the computation. Such a polynomial approximation is the kind of expression that SMM would need to be able to compute efficiently.

3.2 Architectural Design

Given these initial design considerations, the architecture of SMM was configured as in Figure 3.1. The machine has two sections: the computation unit and the microcontroller. The data path specialization of the computation unit and the instruction pipelining of the microcontroller result in maximum utilization of the multiplier/ALU floating point module.

3.2.1 The Computation Unit

The computation unit consists of four parts: the data memory, the register file, the multiplier/ALU module, and the feed-through latch. These sections

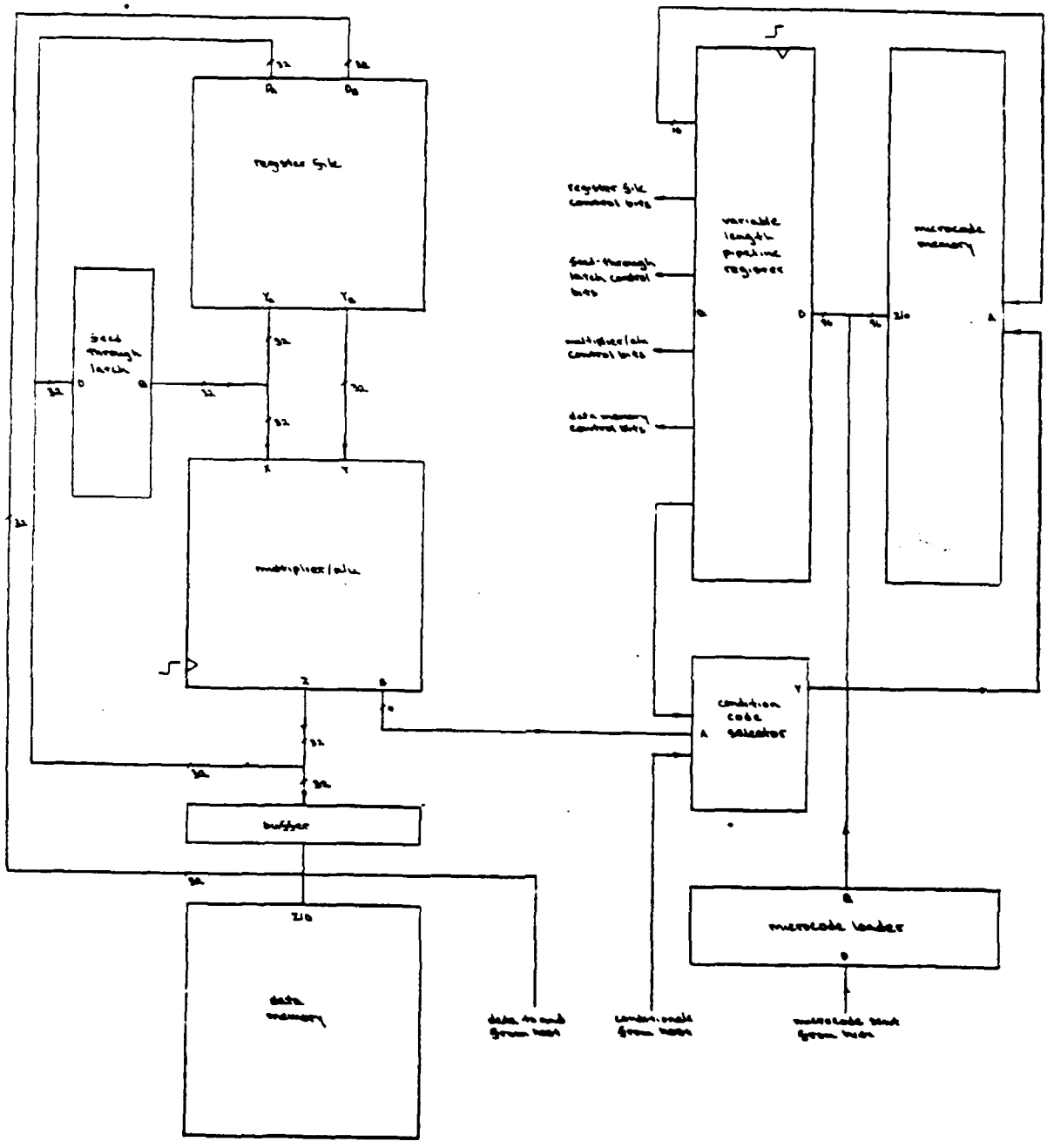

Figure 3.1: Architectural Description

are connected together by 32-bit data paths, providing both single and double precision capability. Data is read from and written to SMM from the host computer via an interface to the data memory system. Values that are to be used for a computation are written to the fast dual-port register file and then fed into the multiplier/ALU unit. The multiplier/ALU component is made up of a Weitek 1264 64-bit IEEE floating point multiplier and a Weitek 1265 64-bit IEEE floating point ALU. Normal operation of the floating point chips is selected to be in pipeline mode to maximize throughput. In this mode, a new 64-bit multiplier operation can be performed every four clock cycles with a latency of ten clock cycles, while a new 64-bit ALU operation can be performed every two clock cycles with a latency of twelve clock cycles. (The minimum clock cycle time for the Weitek chips is 60 nanoseconds.) The results of floating point operations are then sent to the data memory, register file, feed-through latch, or any combination of the three. We try to put the result in the register file if it is to be used in a later operation. The data memory is used if there are currently not enough locations in the register file and the value is not needed immediately. If the result is to be used as an input to the multiplier/ALU module immediately after it has been calculated, it is fed directly to the feed-through latch. The result must be written to data memory if it is to be sent to the host computer.

The feed-through latch is an additional data path feature developed to optimize the computation of polynomial approximations of nonlinear functions such as the sine function. From the series expansion in equation 3.1, we see that such functions are computed serially for a small number of execution units. (If there are a large number of execution units, the polynomial can be rewritten so that the terms can be calculated in parallel, independent of each other.) Because of this, the computation of mappings such as the standard map is a serial computation, since the calculation of both x and y values depends on the value of the sine function. The feed-through latch takes advantage of the serial nature of the calculation by allowing the results of computations flowing out of the floating point unit to be fed directly back into the input of the floating point unit without passing through the register file. Thus, data values can remain in the data paths for repeated computations without ever having to be written back to the register file, reducing the latency of the computation.

Besides numerical data calculations, the Weitek unit is also used to perform

conditional tests, such as comparing the values of two numbers, and converting floating point numbers to integers for data memory write address calculations. Write addresses are calculated such that the low eight bits of the resulting integer are sent to the memory address register and latched in as the high eight bits of a memory write address. This allows fast calculation of page addresses to which to write the results of data computations. Furthermore, the amount of data memory that can be used to write results is not limited by the microcode memory size, as would be the case if write addresses were not calculated but rather were fixed by the microcode instruction. Data memory read addresses are fixed by the microcode instruction because much of the data memory, where the final results of computations are stored until being sent to the host, will not need to be read for the kinds of applications that SMM will run, and direct addressing provides better latency so that data can be read immediately upon execution of the read instruction.

3.2.2 The Microcontroller

The microcontroller consists of a microcode loader, the microcode memory, a variable-length pipeline register, and a condition code selector. Programs for the computer are assembled into microinstructions on the host machine and downloaded from the host through the microcode loader, which then stores the microinstructions into the micromemory. When the microcontroller is operating, the program stored in the microcode memory is executed. The instruction located at the current microaddress is read from the memory and sent to the variable-length pipeline register. The instruction is clocked in through the register, resulting in the proper control signals being sent to the computation unit as well as the microaddress of the next instruction being sent to microcode memory. The length of the pipeline is different for each control signal so that each of the signals coming out of the microcontroller arrives at the appropriate module at the right time. In addition, the length of the entire instruction pipeline is varied by the microinstruction, depending upon whether the multiplier or ALU is being activated. This variation is necessary because the pipelined latency of the multiplier differs from that of the ALU. The condition code selector sends branching instructions to the microcode memory depending on the current state of the machine and the result of conditionals computed by the computation unit. Thus, data

dependent instructions are permitted.

A 96-bit instruction word length was chosen. The function of each instruction bit is described in Appendix A. The long word instruction format allows parallel execution of the subsystems of the computation unit. Instruction flow is as shown in Figure 3.2. One instruction word can be executed

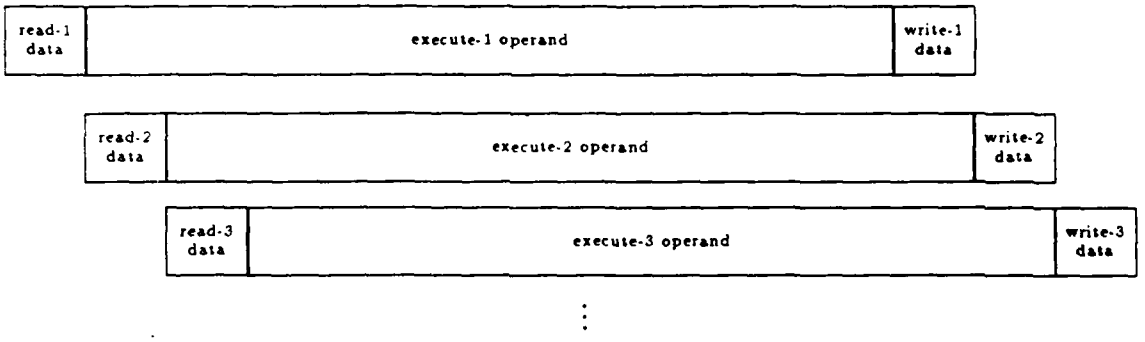

Figure 3.2: Model of Instruction Flow

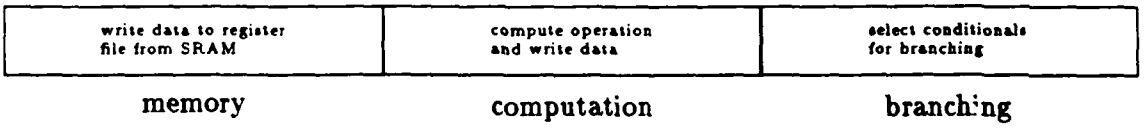

Figure 3.3: Functionality of an Instruction Word

every clock cycle. Since each word can perform a floating point operation, the microcontroller can send out a new floating point operation every clock cycle. This performance is possible because of the variable-length hardware pipeline register. As mentioned above, the microcode dynamically alters the

length of the pipeline so that the control signals arrive to the appropriate subsystem modules at exactly the right time. Furthermore, the hardware pipelining easily facilitates multiple branching by guaranteeing that the write address for the result of the data operation will not disappear until the result has been written, despite the interleaving of instructions.

Each instruction word can be described functionally as shown in Figure 3.3. The memory field is used to write a data value stored in data memory to the register file. The computation field is used to perform an operation using the multiplier/ALU module. The branching field is used for conditional branching.

3.3 Implementation

The design implementation uses standard off-the-shelf parts, primarily Advanced Schottky TTL technology. Single-port SRAM is used for both the data memory system and the micromemory system. There are 2 K words of micromemory locations, 8 K words of data memory locations, and 64 words of register file locations, where an instruction word is 96 bits in length and a data word is 32 bits in length. Four-level pipeline registers that combinatorially select which pipeline level the resulting output comes from are used to implement the microcontroller pipeline. The system clock, write pulses, and latch pulses are all derived using a single delay line. Schematics are given in Appendix B.

3.3.1 Timing

Timing analysis of the machine is shown in Appendix B. Positive edge timing is used. Maximum and minimum propagation delays are taken into account, as well as the tolerances on each tap of the delay line. The minimum clock cycle time is 62.5 nanoseconds as limited by the critical data path latency from the output of the Weitek unit through the feed-through latch back to the input of the Weitek unit. As a result, a 70 nanosecond delay line was chosen for the initial implementation. Reading of data is done on the earlier section of the clock cycle and writing of data is done on the later section.

3.3.2 Construction

The computer was built on a 366 mm x 220 mm wire-wrap board with ground and voltage planes to reduce noise problems. A picture of the completed hardware is shown in Figure 3.4. $.1\mu F$ bypass capacitors were soldered directly between Vcc and GND on dual-in-line chip packages and $47\mu F$ electrolytic bypass capacitors were distributed all over the board. In addition, $4.7\mu F$ tantalum capacitors were connected between Vcc and GND near the register files and Weitek chips because of their extra sensitivity to noise. Clock signals were fed through multiple buffers to avoid fanout and ground bounce problems. Twisted pairs were used for some of the longer wires to reduce noise. Several of the wires required termination because of heavy undershoot resulting from transmission line bounce [8]. Some of the wires which were connected to CMOS inputs were also attached to TTL inputs to reduce undershoot by taking advantage of TTL input clamping diodes. The current implementation uses a 100 nanosecond delay line to produce the system clock, resulting in a 100 nanosecond clock cycle time. Some minor adjustments of the wiring that have not yet been done at this time would reduce the transmission line noise that is present and allow a faster cycle time.

3.4 Use

3.4.1 Host Interface

The Standard Map Machine was implemented to communicate with an HP 9000 Series 300 Computer system [5]. The HP 98630 Breadboard Interface was used to construct memory map hardware to communicate between the backend processor and the host machine. The asynchronous nature of the protocol requires that the processor be in a waiting state or that it be stopped in order for data to be transmitted to and from the processor and the host.

The waiting state is a state, during which the SMM clock is still running, in which SMM promises not to access the data memory, so that the host machine can access the memory safely without worry of bus contention with

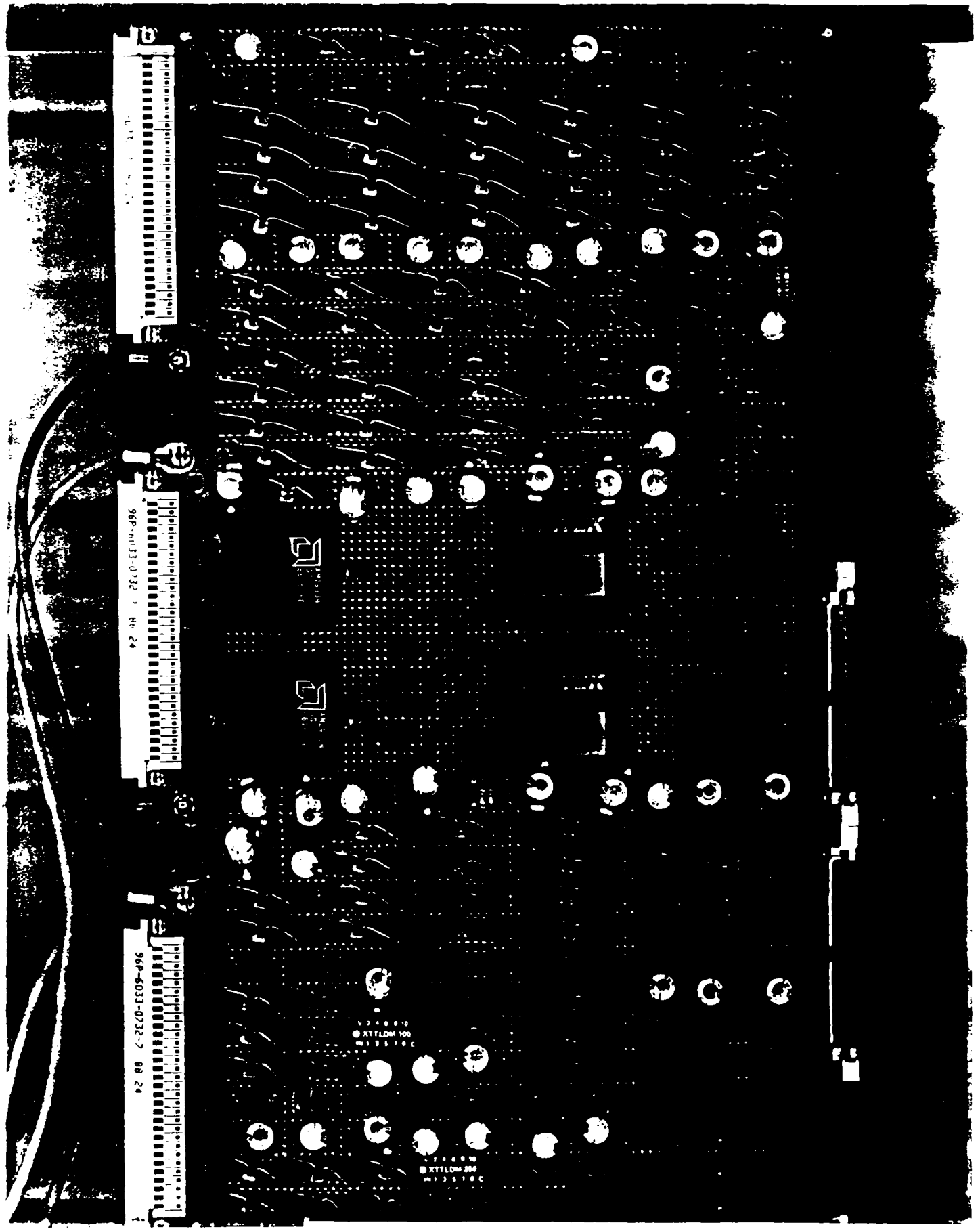

Figure 3.4: The Standard Map Machine

SMM. This state can be written into a program so that data can be sent to and from the host during the execution of a program. Another signal is used to tell SMM when the host is done accessing the SMM memory so that SMM knows when it is safe to jump out of the waiting state and continue processing.

Another way that the host can read and write data to SMM is to send an interrupt signal to SMM, stopping its system clock. In this state, the machine is no longer running and so data can be transmitted between SMM and the host. In addition, programs can be loaded into the microcode memory since the microcontroller has stopped running. Note that programs cannot be loaded if SMM is running (i.e. the system clock is on). To start SMM, a start signal is sent to SMM, which sends the starting instruction word from the microcode loader to the instruction register and enables the SMM clock.

Software was written to allow the user to interface with SMM in Scheme. Commands that are available to the user are listed in the Appendix E. These commands give the user great flexibility in the operation of the machine as well as easy access to the data memory system where the results of computations are stored.

3.4.2 Programming the Machine

An assembler was written by Brian LaMacchia that converts register-transfer code to the proper instruction bits. A detailed explanation of the assembler operation is found in Appendix F. Although there is currently no software that allows the user to program in a high-level language, it is believed that existing software would not be difficult to modify to allow such a programming environment.

3.5 Performance

The inherent maximum performance of any machine using the Weitek 1264/65 chip set is 4 MFlops for a multiplier operation, clocking the machine with the minimum clock cycle time of 60 nanoseconds. The maximum performance on

SMM running at the maximum clock rate of 16 MHz is approximately equal to this absolute maximum. Actual performance of this computer for the class of problems it was designed to solve is about 4 MFlops and is limited by the throughput of the multiplier due to the serial nature of the computation. The current hardware is running at a slower clock rate of 10 MHz, thus giving a performance of about 2.5 MFlops for the single board machine. In addition, since almost all of the computations are performed on the board itself apart from the host machine, these performance levels are sustainable for long periods of time, unlike the floating point accelerators and math coprocessors that are commonly used to speed up calculations in conventional machines.

This high performance is achieved by the specialization of the data paths to numerical computations as well as the specialization of the data path topology to the serial computational nature of the problems that the computer was designed to solve. Furthermore, interleaving of instructions is maximized by the use of variable-length hardware pipelining, thus maximizing the possible throughput of the machine. The overall advantage in the SMM design is that SMM can take large portions of real problems, in the form of microcode programs, and quickly sequence through the given set of instructions to perform numerous computations without ever needing the host to control the flow of data on the board during the computation.

With respect to computations of mappings such as the standard map, the pipelined operation of SMM allows simultaneous calculation of up to three mappings. This is achieved by using the natural multiplier pipeline length of three operations. This parallelism is well suited to studying the behavior of two-dimensional mappings because it allows us to watch the divergence of nearby trajectories by integrating three trajectories simultaneously.

Chapter 4

Studying the Long-term Behavior of the Standard Map

There are a number of interesting applications in which SMM would be useful as a high-performance cost-effective instrument for studying nonlinear dynamics. One problem of interest is studying the long term behavior of the standard map. We would like to determine whether or not the behavior is chaotic for a given set of initial conditions. Chaotic orbits are very sensitive to initial conditions - small changes in the conditions can produce drastically different results. In order to measure the degree of sensitivity to initial conditions and thus determine whether an orbit is chaotic, we estimate the average Lyapunov exponent.

The average Lyapunov exponent is a measure of the exponential rate at which two nearby trajectories diverge, and hence it is a useful way of determining whether or not an orbit is chaotic. The exponent is defined to be

$$\lambda = \lim_{t \to \infty} \frac{\ln \frac{d}{d_0}}{t - t_0} \tag{4.1}$$

where d is the distance between the trajectory and an infinitesimally nearby test trajectory, and t is the number of iterations. A positive Lyapunov exponent is an indication of exponential divergence, and thus chaotic behavior. For an n dimensional system, there can be up to n distinct exponents. These

exponents are paired in Hamiltonian systems - for each nonnegative exponent, there is a corresponding nonpositive exponent of the same magnitude. This implies that for an m degrees of freedom Hamiltonian system, there can be at most m positive exponents [10].

The standard map is a two-dimensional Hamiltonian system. Its area-preserving nature can be seen by noting that the Jacobian for the mapping is equal to one. Since it is a one degree of freedom system, it has at most one positive Lyapunov exponent. If there is no positive exponent, then both exponents are zero and the behavior of the trajectory is regular. If there is a positive exponent, then the behavior is chaotic.

We estimate the largest Lyapunov exponent for various trajectories of the standard map in order to determine whether or not the trajectories exhibit chaotic behavior. This exponent must be either positive, indicating chaotic behavior, or zero, indicating regular behavior. It is not possible for the largest exponent to be negative since the standard map is a Hamiltonian system. A negative exponent is an indication of contraction in the phase-space, but since the system is area-preserving, there must be a direction in which the phase-space is expanding so that the area is preserved. In this direction, the exponent must be positive indicating the expansion of the phase-space in that direction.

We compute the estimation of the largest Lyapunov exponent of a reference trajectory by iterating the variational equations along with the trajectory, thereby linearizing the system in the neighborhood of the reference trajectory and measuring the local divergence of nearby trajectories. The variational equations for the standard map are given in equations 4.2 and 4.3.

$$dx_{n+1} = dx_n + dy_{n+1} \tag{4.2}$$

$$dy_{y+1} = dy_n + k\cos(x_n)dx_n \tag{4.3}$$

From the variations, the distance d between nearby trajectories is given by

$$d = \sqrt{dx^2 + dy^2} \tag{4.4}$$

The exponent can then be calculated using equation 4.1, where t is equal to n. The limit is approximated by averaging the resulting local Lyapunov

(x_0, y_0)	k	iterations	Lyapunov exponent
(.05, .05)	0.2	503407000	between 0 and 4.14e-8
(.05, .05)	0.6	1069280000	.073
(.05, .05)	1.0	551254000	.105
(.05, .05)	2.0	43459500	.436
(1.0, 1.0)	0.2	890956000	between 0 and 2.06e-8
(1.0, 1.0)	0.6	554457000	between 0 and 3.51e-8
(1.0, 1.0)	1.0	709709000	between 0 and 3.14e-8
(1.0, 1.0)	2.0	81642500	.437

Table 4.1: Estimation of Lyapunov exponents of the Standard Map

exponents over long numbers of iterations to get an accurate estimation of the global exponent for that orbit. Periodic renormalization of the variations after several hundred iterations is performed due to the exponential growth of the distance for chaotic orbits. This can be simply done because of the linear nature of the variational equations. The Scheme program and SMM assembler code used to do these calculations are found in Appendix D. The initial variation is chosen as $dx = 1.0$ and $dy = 0.0$. Its orthogonal counterpart is not also calculated because experience has shown that both vectors almost always converge together, after enough iterations, to point in the direction of the greatest rate of expansion of the phase-space. The rare exception occurs when the initial variation points exactly in the direction of fastest contraction, in which case the resulting Lyapunov exponent will be negative, so that it will be clear that the positive exponent is the negative of that result.

Two trajectories were briefly studied. Although it is not possible to generalize from so few cases, it is interesting to examine what the results may suggest. The first trajectory started with the initial conditions $x_0 = .05$, $y_0 = .05$. The second trajectory started with the initial conditions $x_0 = 1.0$, $y_0 = 1.0$. The results as computed for various values of k are given in table 4.1, along with the number of iterations performed.

Preliminary analysis of the two trajectories show that the value of the Lya-

punov exponent depends on the initial conditions and the value of the parameter k. Notice the difference in the exponent for the two trajectories for the same value of k. and notice the difference in the exponent for the same initial conditions for different values of k. However, for larger values of k, the dependence on the initial conditions seems to be lessened, as would be expected by observing the phase-space portrait that was shown in Figure 2.4. This can be seen by comparing the Lyapunov exponents for the two trajectories considered for $k = 2.0$. For small values of k, the dependence of the Lyapunov exponent on initial conditions is again reduced because most of the orbits are then very regular, as indicated in these cases by the very small exponent that approaches zero. The exponent grows with larger values of k, indicating a greater exponential divergence of nearby trajectories. Furthermore, the rate of convergence of the Lyapunov exponent to its limiting value depends on how quickly the phase-space is expanding, which is equivalent to how chaotic the orbit behaves. The faster the rate of expansion, the faster the rate of convergence. Conversely, slower rates of expansion result in slower rates of convergence. The possibility of the Lyapunov exponent not converging is not seen in these trajectories, as data values regularly sampled over the range of iterations indicate a gradual convergence to a single value.

Chapter 5

Past, Present, and Future Work

Although the Standard Map Machine performs well on the class of problems that it was designed to solve, there are a number of flaws in the design of the system, some of which require only minor modifications while others would require a major redesign. In construction, the clock wires were cut to be of equal length rather than to be as short as possible, resulting in transmission line problems due to the length of the wires. This is much more of a problem than clock skew given the size of the board, and hence wires should have been made as short as possible. The multiplier and ALU are never used simultaneously and so can share control lines to reduce the necessary instruction word width. The instruction format is lacking in a constant field, which means that all constants must be loaded into data memory and cannot be placed directly in the instruction stream. The condition code selector currently can only select one bit of the four Weitek status bits. It should be replaced by a PAL to allow all four status bits from the Weitek chips to be read as a condition word rather than four condition bits. Register file locations and microcode memory locations are not directly readable by the host machine, which made debugging more difficult. A more serious problem is that there is no efficient way to take the reciprocal of a number. The 1264/65 Weitek chip set has no functionality for such operations. Since there is no way to generate a good initial guess for the Newton's method (a good guess would

be the number with its exponent negated), a table of initial guesses would have to be stored in data memory to perform division. There is a newer Weitek chip set that performs division and is supposedly compatible with the 1264/65 floating point units, but their specifications have not yet been examined.

At present, some of the problems with the machine mentioned above should be corrected. Furthermore, minor adjustments, such as termination of various wires, can be made to allow a faster cycle time. Investigating the possibility of using the newer Weitek floating point units could enhance the functionality of SMM. Concerning software, it would be desirable to implement a compiler for SMM that would allow programming in a higher level language, to give the user a more familiar programming methodology than the one that is used in the language of the assembler.

From the experience designing SMM for examining chaotic behavior, a number of ideas for future work have come to mind. The first thing is given that we now have such a machine, it is now possible to effectively investigate the long-term behavior of nonlinear systems such as the standard map. The problem mentioned in section 2 would be an intriguing question to research using SMM. The results of such a study would shed light upon an analogous question in a real physical system concerning the orbits of the satellites of Uranus. Another direction of research that can be pursued is in the direction of memory management in computer architectures. Although SMM is fast, the number of data points that it can send to the host is eventually bounded by the rate at which the host can store the data. This is fine for computing such long-term behavior as Lyapunov exponents, in which many computations are required for relatively few resulting data values, but for problems such as plotting the phase-space portrait of the standard map in real time, many data points are required and the speed of the host becomes the limiting factor. It seems that the solution to this bottleneck will become more and more necessary as greater processing power becomes available.

Chapter 6

Conclusion

Studying chaotic behavior in nonlinear systems requires numerous calculations in order to simulate the behavior of such systems. The Standard Map Machine is a special computer designed and implemented for the purpose of performing these intensive computations, such as the computation of the standard map for millions of iterations. The prototype implementation fits on a small wire-wrap board and provides 2.5 MFlops of double-precision computing power for the class of problems that it was designed to solve. Its high-speed and high-precision performance are due to its simple architecture specialized to the numerical computations required of nonlinear systems.

This backend computer has numerous advantages over conventional floating point accelerators and math coprocessors in that almost all of the computations can be performed on SMM itself using its own fast microcontroller rather than relying on the slower instruction control of the host machine, thereby reducing the communication costs between the host and the backend processor that heavily reduce the performance of other machines. Furthermore, unlike costly supercomputing power, SMM serves as a cost-effective instrument that can be completely dedicated for long periods of time as a resource for our numerical simulations of nonlinear systems.

As technology improves, we claim that machines of a similar nature can be designed and implemented as effective instruments for scientific computation. 1988 chip technology can already provide at least twice the performance of

the floating point multiplier/ALU chip set used in SMM. The simpler architecture of such special-purpose computers, tailored to numerical calculations, would allow greater optimization of operations than is feasible in a more general-purpose machine, thereby providing the edge in high-speed and high-precision performance necessary for intensive numerical computations.

Appendix A

Control Bits of the SMM Instruction Word

The following table shows the control fields of the SMM instruction word. The first column gives the bits of the instruction word for the particular control field. with the ordering of the control field bits being the same as that of the instruction word. For instance. bit 73 of the instruction word represents bit 0 of the condition code select control field. The second table column gives the function of each control field and the third column gives the possible range of values of each control field. The 1264/65 function bits, load control bits. and unload control bits are described in detail in the Weitek specifications.

control bits	functional description of bits	possible values for control field
0 - 9	microaddress bits 1-10	0 - 1023
10	wait	0 for not waiting, 1 for waiting
11	regfile B-port write-enable	0 for not enable, 1 for enable
12	regfile A-port latch-enable	0 for not enable, 1 for enable
13	regfile B-port latch-enable	0 for not enable, 1 for enable
14 - 19	regfile A-port read-address	0 - 63
20	regfile A-port/latch output	0 for regfile, 1 for latch
21 - 26	regfile B-port write-address	0 - 63
27 - 32	regfile B-port read-address	0 - 63
33	regfile B-port output	0 for enable, 1 for not enable
34 - 39	1264 multipler function	0 - 63, see Weitek spec.
40 - 45	1265 alu function	0 - 63, see Weitek spec.
46	1264 load enable	0 for enable, 1 for not enable
47	1265 load enable	0 for enable, 1 for not enable
48 - 52	1264 load control	0 - 31, see Weitek spec.
53 - 57	1265 load control	0 - 31, see Weitek spec.
58 - 60	1265 unload control	0 - 7, see Weitek spec.
61	multiplier/alu select	0 for alu, 1 for multipler
62	regfile A-port write-enable	0 for not enable, 1 for enable
63	feed-through latch pulse	0 for do not latch, 1 for latch
64	SRAM write-enable	0 for not enable, 1 for enable
65	SMAR address latch pulse	0 for not enable, 1 for enable
66 - 71	regfile A-port write-address	0 - 63
72	Weitek-to-SRAM buffer	0 for enable, 1 for not enable
73 - 75	conditon code select A, B, C	select cc0 - cc7
76 - 78	1264 unload control	0 - 7, see Weitek spec.
79 - 90	SRAM address	0 - 4095
91	default condition code	0 or 1, set by assembler
92	condition code latch pulse	0 for not enable, 1 for enable
93 - 95	unused	

Table A.1: Description of SMM Instruction Bits

Appendix B

Design Schematics and Timing Diagrams

The design schematics and timing diagrams for SMM and the host interface are shown in the following figures. Figure B.1 gives a functional description of the overall system with a detailed view of the system clock design showing the clock generator as well as the derived write pulses and latch pulses. Note that the clock buffers that were included in the actual construction to reduce fanout are not shown in the diagram. Figure B.2 gives a detailed diagram of the wire connections in the computation unit. Figure B.3 is a detailed drawing of the microcontroller. There are three bits of the 96-bit instruction word that are currently unused. Figure B.4 shows the hardware for the host interface as well as the timing of interface signals. The signals from the host are transmitted from the breadboard interface card that is plugged into the backplane of the host, through ribbon cables, to the control and data buffers. The breadboard interface is not shown here. Figure B.5 shows the timing analysis of the system.

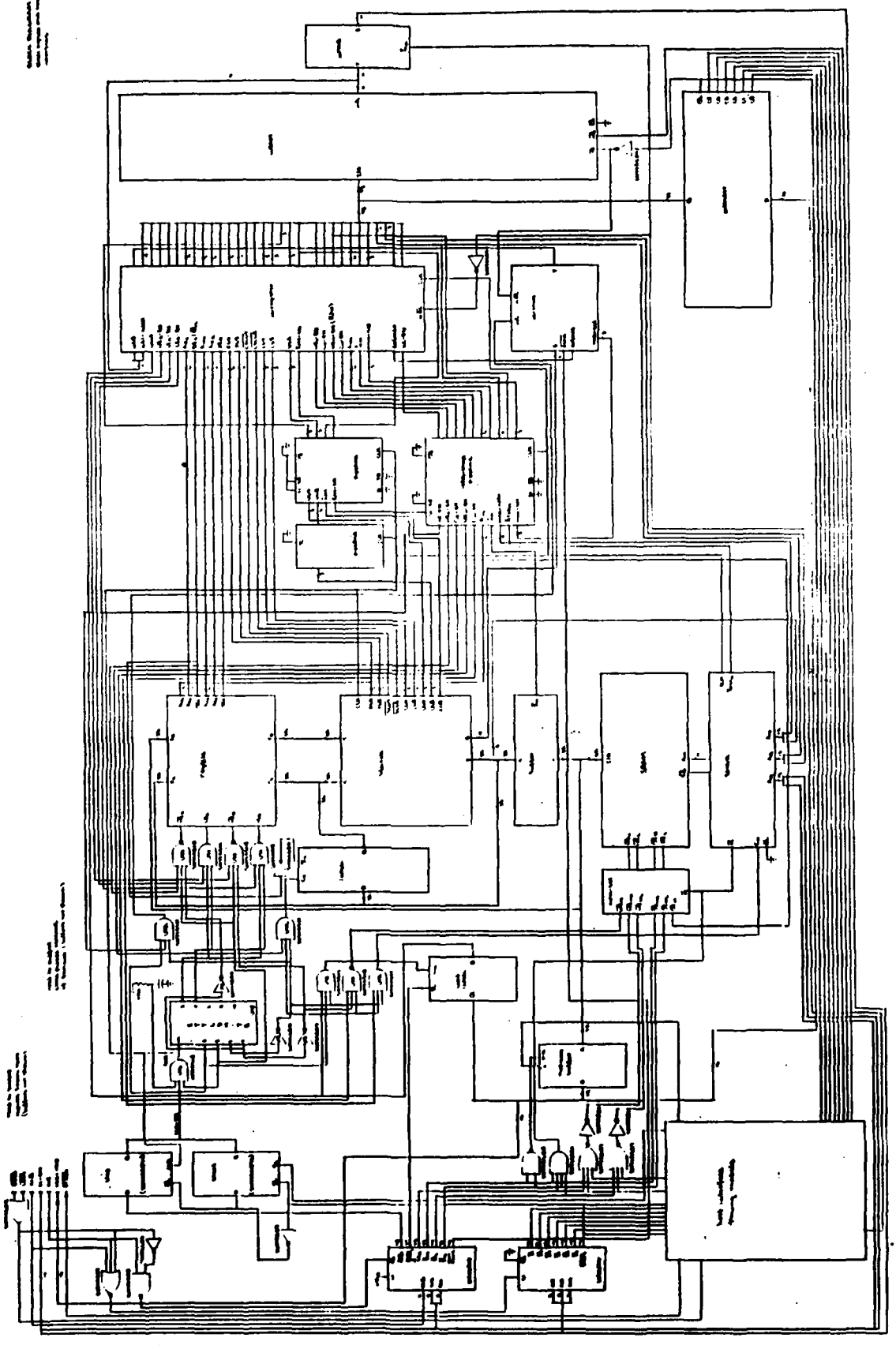

Figure B.1: Functional Description with System Clock

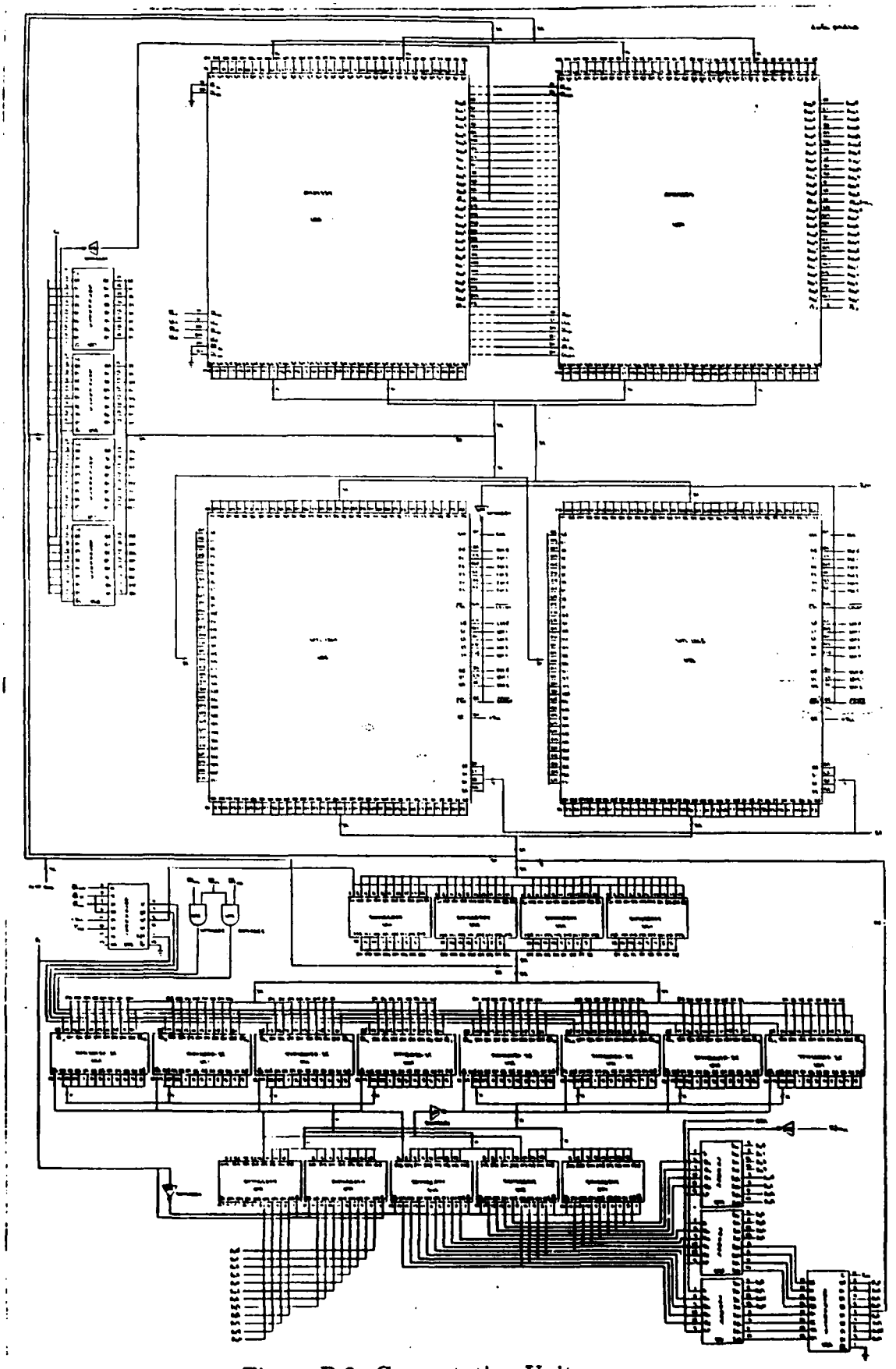

Figure B.2: Computation Unit

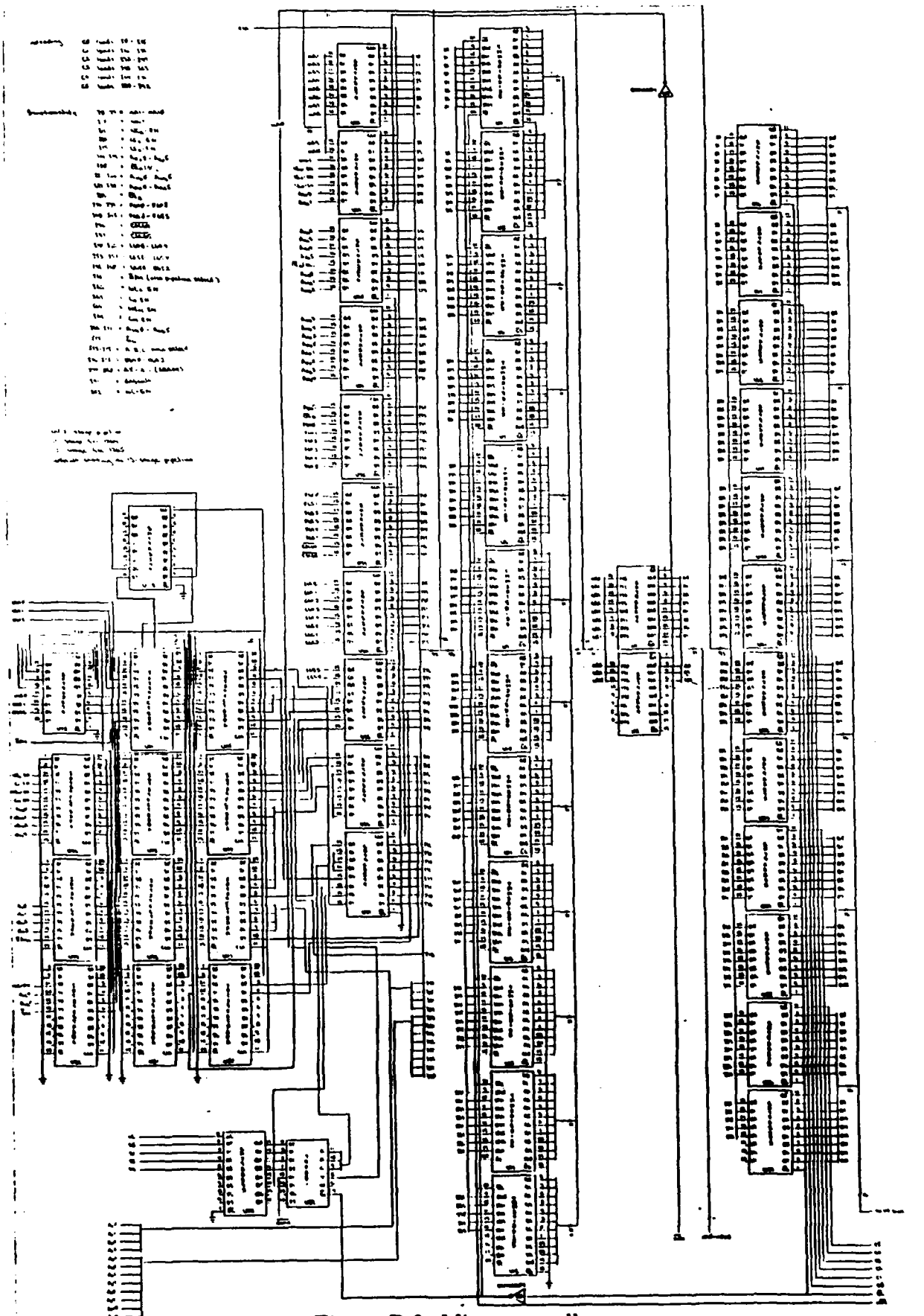

Figure B.3: Microcontroller

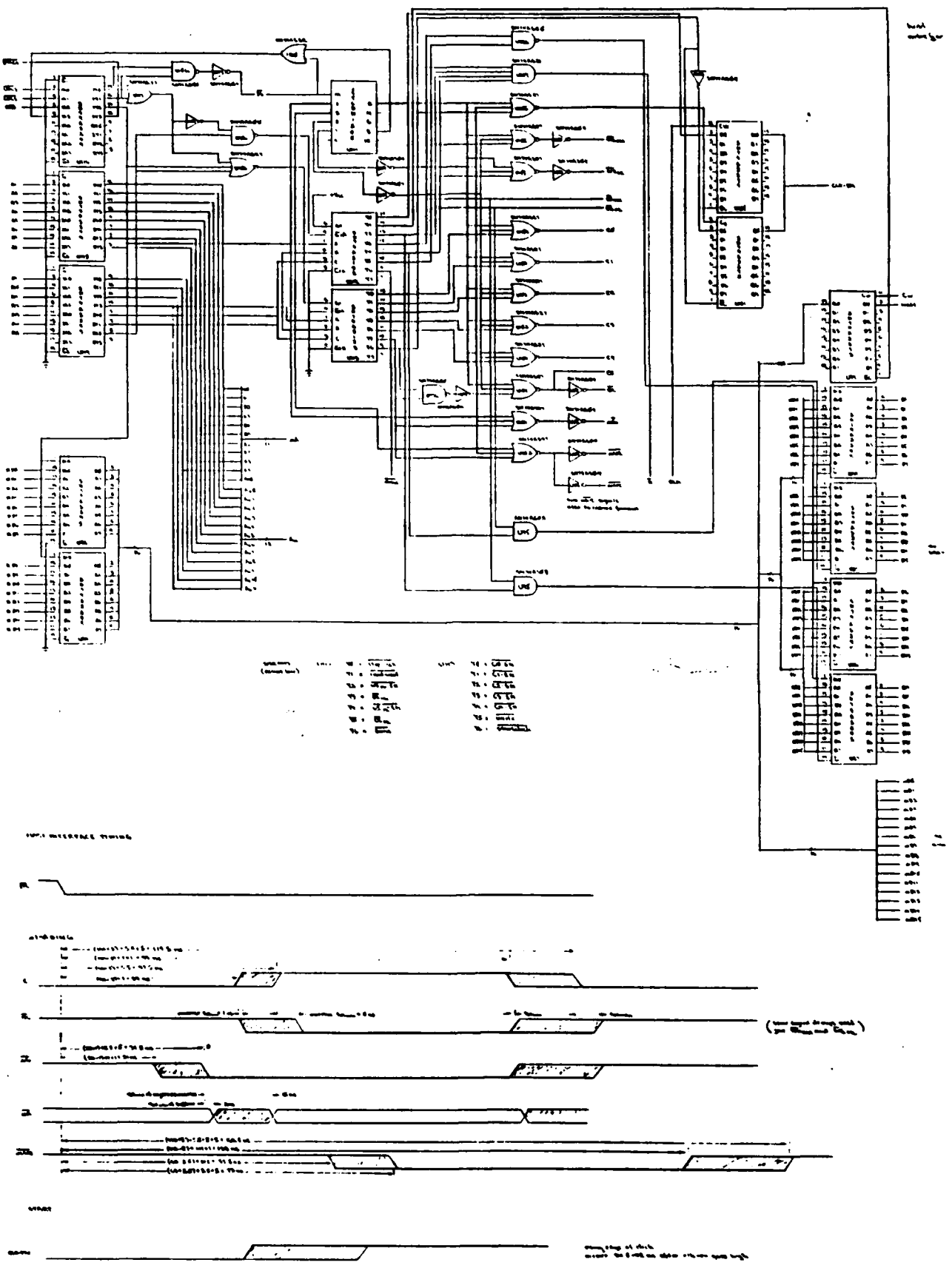

Figure B.4: Host Interface

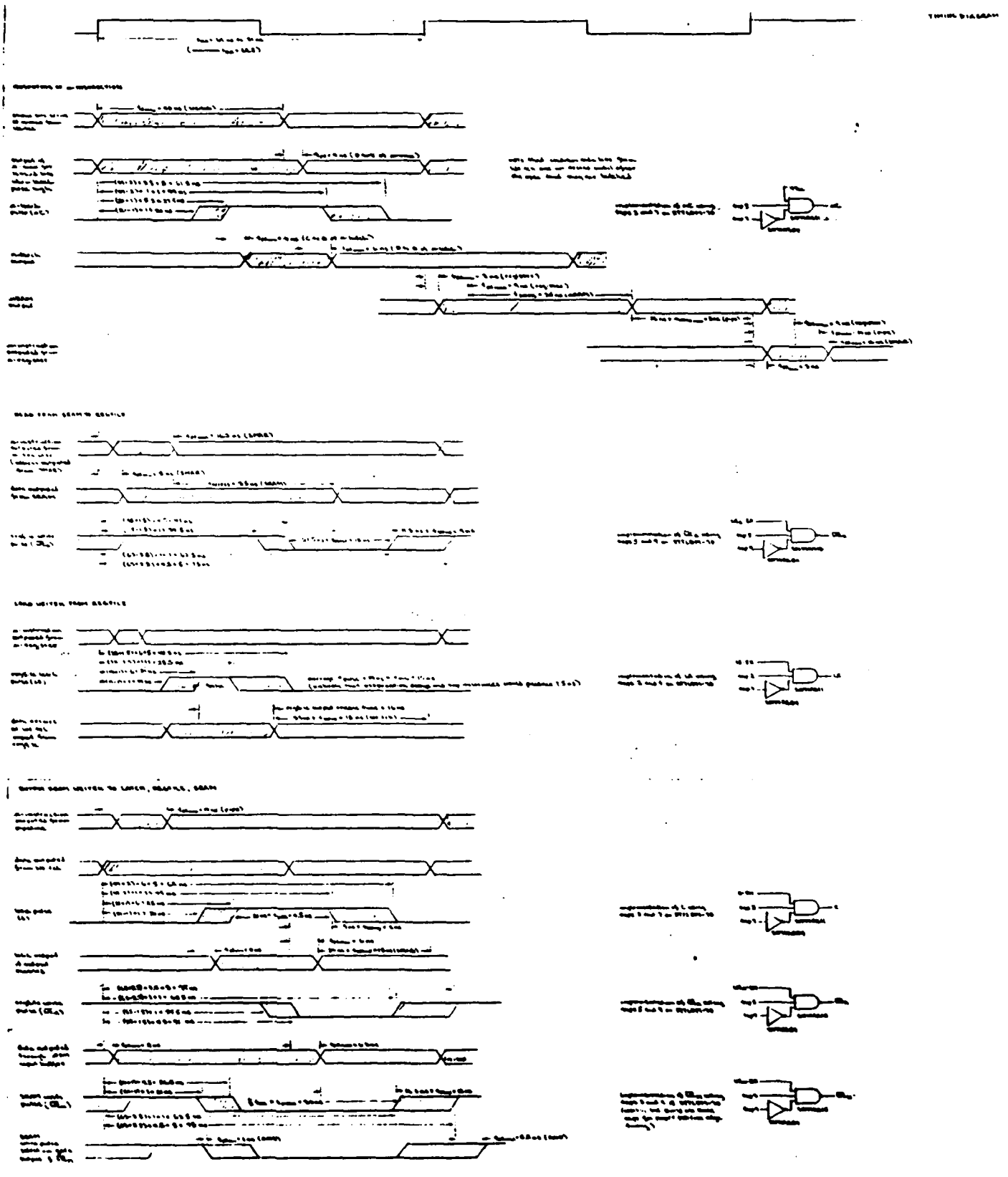

Figure B.5: Timing Diagram

Appendix C

Sine and Cosine Approximations

The sine function was approximated using the following series expansion:

$$\begin{aligned}\sin(x) &= x(s_1 + x^2(s_3 + x^2(s_5 + \\ &\quad x^2(s_7 + x^2(s_9 + x^2(s_{11} + s_{13}x^2))))))\end{aligned} \tag{C.1}$$

where the values of the coefficients were computed to be

$$s_1 = \frac{97429332733154293903}{9742933273315429 6875}$$

$$s_3 = -\frac{433019256591792251}{2598115539550781250}$$

$$s_5 = \frac{494879150388841}{59385498046875000}$$

$$s_7 = -\frac{46142578073}{232558593750000}$$

$$s_9 = \frac{1599608879}{580466250000000}$$

$$s_{11} = -\frac{63979907}{2554051500000000}$$

$$s_{13} = \frac{64879}{408648240000000}$$

The cosine function was approximated using a similar series expansion:

$$\begin{aligned}\cos(x) &= c_0 + x^2(c_2 + x^2(c_4 + \\ &\quad x^2(c_6 + x^2(c_8 + x^2(c_{10} + c_{12}x^2))))))\end{aligned} \tag{C.2}$$

where the values of the coefficients were computed to be

$$c_1 = \frac{65608978271484371}{65608978271484375}$$

$$c_3 = -\frac{259811553955073273}{519623107910156250}$$

$$c_5 = \frac{22840576171747}{548173828125000}$$

$$c_7 = -\frac{733154295719}{527871093750000}$$

$$c_9 = \frac{3519139237}{141891750000000}$$

$$c_{11} = -\frac{12795833}{46437300000000}$$

$$c_{13} = \frac{56179}{27243216000000}$$

Both expansions are accurate to 64-bit IEEE double precision full accuracy for values of x in the interval $[0, \pi/4]$. The coefficients are expressed as rational numbers for maximum accuracy. Values of x outside of this range are converted to equivalent the sine or cosine function whose argument falls within this range. Note that for the standard map, $0 < x < 2\pi$. A table of conversions for the range of values of the argument to the sine function in the standard map is given in the following table.

Range	Sine	Cosine
$0 \leq x \leq \frac{\pi}{4}$	$\sin(x)$	$\cos(x)$
$\frac{\pi}{4} < x \leq \frac{\pi}{2}$	$\cos(\frac{\pi}{2} - x)$	$\sin(\frac{\pi}{2} - x)$
$\frac{\pi}{2} < x \leq \frac{3\pi}{4}$	$\cos(x - \frac{\pi}{2})$	$-\sin(x - \frac{\pi}{2})$
$\frac{3\pi}{4} < x \leq \pi$	$\sin(\pi - x)$	$-\cos(\pi - x)$
$\pi < x \leq \frac{5\pi}{4}$	$-\sin(x - \pi)$	$-\cos(x - \pi)$
$\frac{5\pi}{4} < x \leq \frac{3\pi}{2}$	$-\cos(\frac{3\pi}{2} - x)$	$-\sin(\frac{3\pi}{2} - x)$
$\frac{3\pi}{2} < x \leq \frac{7\pi}{4}$	$-\cos(x - \frac{3\pi}{2})$	$\sin(x - \frac{3\pi}{2})$
$\frac{7\pi}{4} < x < 2\pi$	$-\sin(2\pi - x)$	$\cos(2\pi - x)$

Table C.1: Sine and Cosine Conversions for Values of x

Appendix D

Program for Calculating Lyapunov Exponents

The following Scheme programs was used to interface to SMM while it was running the Lyapunov exponents program:

```
(define (wait-loop)
  (if (even? (wait))
      (wait-loop)))

(define init 5)

(define (lyp k x y renorm)
  (define dx 1.0)
  (define dy 0.0)
  (define t 0)
  (define twopi (* 8 (atan 1)))
  (define twopi-1 (/ 1 (* 8 (atan 1))))
  (define sign 0)
  (define s1 (/ 97429332733154293903 97429332733154296875))
  (define s3 (/ -433019256591792251 2598115539550781250))
```

```
(define s5 (/ 494879150388841 59385498046875000))
(define s7 (/ -46142578073 232558593750000))
(define s9 (/ 1599608879 580466250000000))
(define s11 (/ -63979907 2554051500000000))
(define s13 (/ 64879 408648240000000))
(define c0 (/ 65608978271484371 65608978271484375))
(define c2 (/ -259811553955073273 519623107910156250))
(define c4 (/ 22840576171747 548173828125000))
(define c6 (/ -733154295719 527871093750000))
(define c8 (/ 3519139237 141891750000000))
(define c10 (/ -12795833 46437300000000))
(define c12 (/ 56179 27243216000000))
(define one 1)
(define i 0)
(define pi (* 4 (atan 1)))
(define xpif (* 1.5 pi))
(define fpif (* 1.25 pi))
(define tpif (* .75 pi))
(define pit (* .5 pi))
(define pif (* .25 pi))
(define spif (* 1.75 pi))
(define d 1)
(define logd 0)
(define (run-loop)
  (if (even? (wait))
      (run-loop)
      (begin
        (set! t (upload-data 92))
        (set! d (sqrt (upload-data 80)))
        (set! x (upload-data 84))
        (set! y (upload-data 86))
        (set! dx (/ (upload-data 88) d))
        (set! dy (/ (upload-data 90) d))
        (download-data `((88 ,dx) (,dy)))
        (done)
        (set! logd (+ logd (log d)))
        (display t)
        (display "  ")
        (display (/ logd t))
```

```
          (display "  ")
          (display x)
          (display "  ")
          (display y)
          (newline)
          (run-loop))))
  (chasm-file "lyp-board")
  (download-code *chasm-bits*)
  (start-board 0)
  (wait-loop)
  (download-data `((82 ,k) (,x) (,y) (,dx) (,dy) (,t) (,renorm)))
  (download-data `((96 ,twopi-1) (,twopi)))
  (download-data `((100 ,s1) (,s3) (,s5) (,s7) (,s9) (,s11) (,s13)))
  (download-data `((114 ,c0) (,c2) (,c4) (,c6) (,c8) (,c10) (,c12)))
  (download-data `((128 ,one) (,i) (,sign)))
  (download-data `((134 ,xpif) (,fpif) (,pi) (,tpif) (,pit) (,pif) (,spif)))
  (download-data `((148 ,init)))
  (done)
  (newline)
  (display "start with k, x, y, renorm values:  ")
  (display k)
  (display ", ")
  (display x)
  (display ", ")
  (display y)
  (display ", ")
  (display renorm)
  (newline)
  (run-loop))

(define (slyp k x y renorm)
  (newline)
  (stop-clock!)
  (define dx 1.0)
  (define dy 0.0)
  (define t 0)
  (define logd 0)
  (define d)
  (define (run-loop)
```

```
 (if (even? (wait))
     (run-loop)
     (begin
       (set! t (upload-data 92))
       (set! d (sqrt (upload-data 80)))
       (set! x (upload-data 84))
       (set! y (upload-data 86))
       (set! dx (/ (upload-data 88) d))
       (set! dy (/ (upload-data 90) d))
       (download-data `((88 ,dx) (,dy)))
       (done)
       (set! logd (+ logd (log d)))
       (display t)
       (display "  ")
       (display (/ logd t))
       (display "  ")
       (display x)
       (display "  ")
       (display y)
       (newline)
       (run-loop))))
(start-board 0)
(wait-loop)
(download-data `((82 ,k) (,x) (,y) (,dx) (,dy) (,t) (,renorm)))
(done)
(display "start with k, x, y, renorm values:  ")
(display k)
(display ", ")
(display x)
(display ", ")
(display y)
(display ", ")
(display renorm)
(newline)
(run-loop))
```

The procedure **lyp** assembles the program stored in the file "lyp-board.scm", loads the resulting instruction bits into the SMM microcode memory, and stores the appropriate data values into data memory using the appropriate download commands. Note that the program starts SMM and waits until it is in a waiting state before it downloads data to the board, using the procedure **wait-loop**. Data is downloaded to the board after the machine has been started in order to guarantee that the pipeline has been cleared of extraneous old data memory write instructions that may erase the new data that has been stored. SMM then iterates the variational equations with the standard map *renorm* times using the initial values of k, x, and y, waits for the host to pick up the result, and then iterates another *renorm* times with renormalized dx and dy values. The Scheme program processes the data to be displayed, renormalizes values, tells SMM that it is done accessing the data memory so that SMM can proceed, and displays the result. The data is read from the data memory page address given by init. For example, since init is equal to 5 and the value of t is stored by the microcode program, as shown below, in data memory location 12 on that page, data memory address 92 ($16 * 5 + 12$) must be read for the value of t. The program continues to loop until the user interrupts it. **lyp1** does the same thing as **lyp** except that it does not reload the microcode or series approximation coefficients. **lyp1** reduces some of the initial overhead so that the user does not have to wait for the microcode program to be assembled and downloaded every time. It can be used after **lyp** has been used once so that the necessary bits have been loaded into SMM. The assembler program that is used by SMM to calculate the standard map and the Lyapunov exponents program is listed below.

```
;;; this is the standard map program for Lyapunov exponents on SMM

'(
  (clear-wait-bit)              ;;; makes sure wait bit is cleared
  (alu-load-mode #b000000)      ;;; loads proper mode of operation for alu
  (alu-load-mode #b011111)
  (alu-load-mode #b100010)
  (mul-load-mode #b000000)      ;;; loads proper mode of operation for multiplier
  (mul-load-mode #b010110)
  (mul-load-mode #b100100)
```

```
(wait)                          ;;; allows data to be loaded after pipe cleared

(da (r 0) (s 82))               ;;; k
(da (r 2) (s 84))               ;;; x
(da (r 4) (s 86))               ;;; y
(da (r 10) (s 92))              ;;; t
(da (r 12) (s 94))              ;;; n
(da (r 14) (s 96))              ;;; twopi-1
(da (r 16) (s 98))              ;;; twopi
(da (r 18) (s 100))             ;;; s1
(da (r 20) (s 102))             ;;; s3
(da (r 22) (s 104))             ;;; s5
(da (r 24) (s 106))             ;;; s7
(da (r 26) (s 108))             ;;; s9
(da (r 28) (s 110))             ;;; s11
(da (r 30) (s 112))             ;;; s13
(da (r 32) (s 114))             ;;; c0
(da (r 34) (s 116))             ;;; c2
(da (r 36) (s 118))             ;;; c4
(da (r 38) (s 120))             ;;; c6
(da (r 40) (s 122))             ;;; c8
(da (r 42) (s 124))             ;;; c10
(da (r 44) (s 126))             ;;; c12
(da (r 46) (s 128))             ;;; one
(da (r 48) (s 130))             ;;; i
(da (r 50) (s 132))             ;;; sign

(:newloop da (r 6) (s 88))      ;;; dx
(da (r 8) (s 90))               ;;; dy

(:loop da (r 60) (s 148))                       ;;; memory latch offset
(da (none) (f->is (r 60)))                      ;;; latch page address
(da (r 62) (s 140))                             ;;; 3pi/4
(da (r 60) (s 136))                             ;;; 5pi/4
(da (none) (cmp (r 2) (r 58)) (lcc))            ;;; test x > 7pi/4
(da (r 58) (s 134))                             ;;; 6pi/4
(da (none) (cmp (r 2) (r 58)) (lcc))            ;;; test x > 6pi/4
(da (none) (cmp (r 2) (r 60)) (lcc))            ;;; test x > 5pi/4
(da (none) (cmp (r 2) (r 52)) (lcc))            ;;; test x > pi
```

```
(da (none) (cmp (r 2) (r 62)) (lcc))            ;;; test x > 3pi/4
(da (r 62) (- (r 16) (r 2)) (if cc1 :poly7))
(da (none) (cmp (r 2) (r 54)) (lcc))            ;;; test x > pi/2
(da (r 62) (- (r 2) (r 58)) (if cc1 :poly6))
(da (r 62) (- (r 58) (r 2)) (if cc1 :poly5))
(da (r 62) (- (r 2) (r 52)) (if cc1 :poly4))
(da (none) (cmp (r 2) (r 56)) (lcc))            ;;; test x > pi/4
(da (r 62) (- (r 52) (r 2)) (if cc1 :poly3))
(da (r 62) (- (r 2) (r 54)) (if cc1 :poly2))
(nop)
(nop)
(nop)
(nop)
(nop)
(nop)
(da (r 62) (- (r 54) (r 2)) (if cc1 :poly1))

(:poly0 da (r 62) (i (r 2)))
(nop)
(nop)
(da (r 58) (* (r 0) (r 6)) (goto :sin-cos))     ;;; compute k*dx
(:poly1 nop)
(nop)
(da (r 58) (* (r 0) (r 6)) (goto :cos-sin))     ;;; compute k*dx
(:poly2 nop)
(nop)
(da (r 58) (-* (r 0) (r 6)) (goto :cos-sin))    ;;; compute -k*dx
(:poly3 nop)
(nop)
(da (r 58) (-* (r 0) (r 6)) (goto :sin-cos))    ;;; compute -k*dx
(:poly4 nop)
(nop)
(da (r 58) (-* (r 0) (r 6)))                    ;;; compute -k*dx
(da (r 50) (i (r 46)) (goto :-sin-cos))         ;;; set sign to be negative
(:poly5 nop)
(nop)
(da (r 58) (-* (r 0) (r 6)))                    ;;; compute -k*dx
(da (r 50) (i (r 46)) (goto :-cos-sin))         ;;; set sign to be negative
(:poly6 nop)
```

```
(nop)
(da (r 58) (* (r 0) (r 6)))                    ;;; compute k*dx
(da (r 50) (i (r 46)) (goto :-cos-sin))        ;;; set sign to be negative
(:poly7 nop)
(nop)
(da (r 58) (* (r 0) (r 6)))                    ;;; compute k*dx
(da (r 50) (i (r 46)) (goto :-sin-cos))        ;;; set sign to be negative

(:sin-cos nop)
(nop)
(:-sin-cos nop)
(nop)
(nop)
(nop)
(:funct nop)
(nop)
(da (r 60) (* (r 62) (r 62)))                  ;;; compute x^2
(nop)
(nop)
(da (r 56) (* (r 62) (r 18)))                  ;;; compute s1*x
(da (none) (cmp (r 50) (r 46)) (lcc))          ;;; tests sign for plus or minus
(nop)
(nop)
(nop)
(nop)
(da (latch) (* (r 60) (r 30)))                 ;;; compute s13*x^2
(nop)
(nop)
(da (latch) (* (r 60) (r 44)))                 ;;; compute c12*x^2
(nop)
(nop)
(da ((latch) (r 54)) (* (r 62) (r 60)))        ;;; compute x^3
(da (latch) (+ (latch) (r 28)))                ;;; compute s11 + s13*x^2
(nop)
(nop)
(da (latch) (+ (latch) (r 42)))                ;;; compute c10 + c12*x^2
(nop)
(nop)
(da (latch) (* (latch) (r 20)))                ;;; compute s3*x^3
```

```
(nop)
(nop)
(da (latch) (* (latch) (r 60)))          ;;; compute (s11 + s13*x^2) * x^2
(nop)
(nop)
(da (latch) (* (latch) (r 60)))          ;;; compute (c10 + c12*x^2) * x^2
(da (r 56) (+ (latch) (r 56)))           ;;; compute s1*x + s3*x^3
(nop)
(nop)
(da (latch) (+ (latch) (r 26)))          ;;; compute s9 + sin-poly
(nop)
(nop)
(da (latch) (+ (latch) (r 40)))          ;;; compute c8 + cos-poly
(nop)
(nop)
(da (r 54) (* (r 60) (r 54)))            ;;; compute x^5
(nop)
(nop)
(da (latch) (* (latch) (r 60)))          ;;; compute sin-poly * x^2
(nop)
(nop)
(da (latch) (* (latch) (r 60)))          ;;; compute cos-poly * x^2
(nop)
(nop)
(da (latch) (* (r 34) (r 60)))           ;;; compute c2*x^2
(da (latch) (+ (latch) (r 24)))          ;;; compute s7 + sin-poly
(nop)
(nop)
(da (latch) (+ (latch) (r 38)))          ;;; compute c6 + cos-poly
(nop)
(nop)
(da (r 52) (+ (latch) (r 32)))           ;;; compute c0 + c2*x^2
(nop)
(nop)
(da (latch) (* (latch) (r 60)))          ;;; compute sin-poly * x^2
(nop)
(nop)
(da (latch) (* (latch) (r 60)))          ;;; compute cos-poly * x^2
(nop)
```

```
(nop)
(da (r 60) (* (r 60) (r 60)))          ;;; compute x^4
(da (latch) (+ (latch) (r 22)))        ;;; compute s5 + sin-poly
(nop)
(nop)
(da (latch) (+ (latch) (r 36)))        ;;; compute c4 + cos-poly
(nop)
(nop)
(da (r 48) (+ (r 46) (r 48)))          ;;; increment i
(nop)
(nop)
(da (latch) (* (latch) (r 54)))        ;;; compute sin-poly * x^5
(nop)
(nop)
(da (latch) (* (latch) (r 60)))        ;;; compute cos-poly * x^4
(nop)
(nop)
(nop)
(nop)
(da (latch) (+ (latch) (r 56)))        ;;; finish computing sin-poly
(nop)
(nop)
(da (latch) (+ (latch) (r 52)))        ;;; finish computing cos-poly
(nop (goto :result))

(:cos-sin nop)
(nop)
(:-cos-sin nop)
(nop)
(nop)
(nop)
(nop)
(nop)
(da (r 60) (* (r 62) (r 62)))          ;;; compute x^2
(nop)
(nop)
(da (r 56) (* (r 62) (r 18)))          ;;; compute s1*x
(da (none) (cmp (r 50) (r 46)) (lcc))  ;;; tests sign for plus or minus
(nop)
```

```
(nop)
(nop)
(nop)
(da (latch) (* (r 60) (r 44)))              ;;; compute c12*x^2
(nop)
(nop)
(da (latch) (* (r 60) (r 30)))              ;;; compute s13*x^2
(nop)
(nop)
(da ((latch) (r 54)) (* (r 62) (r 60)))     ;;; compute x^3
(da (latch) (+ (latch) (r 42)))             ;;; compute c10 + c12*x^2
(nop)
(nop)
(da (latch) (+ (latch) (r 28)))             ;;; compute s11 + s13*x^2
(nop)
(nop)
(da (latch) (* (latch) (r 20)))             ;;; compute s3*x^3
(nop)
(nop)
(da (latch) (* (latch) (r 60)))             ;;; compute (c10 + c12*x^2) * x^2
(nop)
(nop)
(da (latch) (* (latch) (r 60)))             ;;; compute (s11 + s13*x^2) * x^2
(da (r 56) (+ (latch) (r 56)))              ;;; compute s1*x + s3*x^3
(nop)
(nop)
(da (latch) (+ (latch) (r 40)))             ;;; compute c8 + cos-poly
(nop)
(nop)
(da (latch) (+ (latch) (r 26)))             ;;; compute s9 + sin-poly
(nop)
(nop)
(da (r 54) (* (r 60) (r 54)))               ;;; compute x^5
(nop)
(nop)
(da (latch) (* (latch) (r 60)))             ;;; compute cos-poly * x^2
(nop)
(nop)
(da (latch) (* (latch) (r 60)))             ;;; compute sin-poly * x^2
```

```
(nop)
(nop)
(da (latch) (* (r 34) (r 60)))          ;;; compute c2*x^2
(da (latch) (+ (latch) (r 38)))         ;;; compute c6 + cos-poly
(nop)
(nop)
(da (latch) (+ (latch) (r 24)))         ;;; compute s7 + sin-poly
(nop)
(nop)
(da (r 52) (+ (latch) (r 32)))          ;;; compute c0 + c2*x^2
(nop)
(nop)
(da (latch) (* (latch) (r 60)))         ;;; compute cos-poly * x^2
(nop)
(nop)
(da (latch) (* (latch) (r 60)))         ;;; compute sin-poly * x^2
(nop)
(nop)
(da (r 60) (* (r 60) (r 60)))           ;;; compute x^4
(da (latch) (+ (latch) (r 36)))         ;;; compute c4 + cos-poly
(nop)
(nop)
(da (latch) (+ (latch) (r 22)))         ;;; compute s5 + sin-poly
(nop)
(nop)
(da (r 48) (+ (r 46) (r 48)))           ;;; increment i
(nop)
(nop)
(da (latch) (* (latch) (r 60)))         ;;; compute cos-poly * x^4
(nop)
(nop)
(da (latch) (* (latch) (r 54)))         ;;; compute sin-poly * x^5
(nop)
(nop)
(nop)
(nop)
(da (latch) (+ (latch) (r 52)))         ;;; compute cos-poly for sine
(nop)
(nop)
```

```
(da (latch) (+ (latch) (r 56)))                ;;; compute sin-poly for cosine
(nop (goto :result))

(:result nop)
(da (none) (none) (if cc0 :no-negate))
(nop)
(nop)
(da (latch) (-* (latch) (r 0)) (goto :next)) ;;; compute -k * sin x
(:no-negate nop)
(nop)
(da (latch) (* (latch) (r 0)))                 ;;; compute k * sin x
(:next nop)
(nop)
(da (latch) (* (latch) (r 58)))                ;;; compute +/- k * cos x * dx
(nop)
(nop)
(da (r 50) (- (r 50) (r 50)))                  ;;; set sign to zero
(da ((latch) (r 4)) (+ (latch) (r 4)))         ;;; compute new y value
(nop)
(nop)
(da ((latch) (r 8)) (+ (latch) (r 8)))         ;;; compute new dy value
(nop)
(nop)
(da (none) (cmp (r 48) (r 12)) (lcc))          ;;; test i<n
(nop)
(nop)
(da (r 2) (+ (latch) (r 2)))                   ;;; compute new x value
(nop)
(nop)
(da (r 6) (+ (latch) (r 6)))                   ;;; compute new dx value
(nop)
(nop)
(da (r 62) (* (r 4) (r 14)))                   ;;; compute y / 2pi
(nop)
(nop)
(da (r 58) (* (r 8) (r 8)))                    ;;; compute dy * dy
(nop)
(nop)
(da (latch) (* (r 2) (r 14)))                  ;;; compute x / 2pi
```

```
(alu-load-mode #b001100)                ;;; change rounding mode
(nop)
(da (latch) (f->i (r 62)))              ;;; floor (y / 2pi)
(nop)
(nop)
(da (r 62) (* (r 6) (r 6)))             ;;; compute dx * dx
(da (latch) (f->i (latch)))             ;;; floor (x / 2pi)
(nop)
(nop)
(nop)
(nop)
(da (latch) (i->f (latch)))
(nop)
(nop)
(nop)
(nop)
(da (r 56) (i->f (latch)))
(nop)
(nop)
(nop)
(nop)
(da (r 60) (* (latch) (r 16)))          ;;; floor-y * 2pi
(nop)
(nop)
(da (r 54) (s 142))                     ;;; pi/2
(nop)
(nop)
(da (r 56) (* (r 56) (r 16)))           ;;; floor-x * 2pi
(alu-load-mode #b000000)                ;;; return to normal rounding mode
(nop)
(da (r 60) (+ (r 62) (r 58)))           ;;; compute dx*dx + dy*dy
(da (r 4) (- (r 4) (r 60)))             ;;; calculate new y value with mod 2p
(da (r 52) (s 138))                     ;;; pi
(da (r 58) (s 146))                     ;;; 7pi/4
(da (r 2) (- (r 2) (r 56)))             ;;; calculate new x value with mod 2p
(da (r 56) (s 144))                     ;;; pi/4
(da (none) (none) (if cc0 :loop))       ;;; if i<n, then go back to :loop
(da (none) (cmp (r 60) (r 46)) (lcc))   ;;; test to see if dx*dx + dy*dy > 1
(da ((r 10) (s 12)) (+ (r 10) (r 48)))  ;;; t = t + i
```

```
(da (r 48) (- (r 48) (r 48)))                ;;; clear i
(da (s 0) (i (r 60)))                        ;;; write dx*dx + dy*dy to memory
(da (s 14) (i (r 12)))                       ;;; write n to memory
(da (s 4) (i (r 2)))                         ;;; write x to memory
(da (s 6) (i (r 4)))                         ;;; write y to memory
(da (s 8) (i (r 6)))                         ;;; write dx to memory
(da (s 10) (i (r 8)))                        ;;; write dy to memory
(da (none) (none) (if cc1 :update-loop))     ;;; if > then save values
(nop)
(nop)
(nop)                                        ;;; nops needed to guarantee
(nop)                                        ;;; writes to data memory completed
(nop)
(nop)
(nop)
(nop)
(da (none) (none) (goto :loop))
(:update-loop nop)
(wait)
(da (none) (none) (goto :newloop))
)
```

There are a number of places in the assembler code for computing the Lyapunov exponents where there are unused instruction cycles consisting of several consecutive nop instructions. Additional operations can be performed during these instruction cycles if desired. Note that in order to calculate the floor function for modulo 2π, the rounding mode of the alu is changed in the middle of the program from the default rounding mode of rounding to nearest significand to the rounding mode of rounding toward negative infinity, according to the information provided in the Weitek specifications. This program only calculates one trajectory of the standard map. Although there is a great deal of instruction interleaving present, the trigonometric functions are not calculated using the minimum number of operations in order to make use of some unused instruction cycles. More efficient use of the machine can be made by computing three trajectories simultaneously.

Appendix E

Interface Commands

The address space of the backend computer is mapped into the address space of the HP Series 200 computer system that serves as the host. Scheme procedures have been defined that can read and write 16-bit words to address *locations of the board:*

```
;; rc takes an address value between #b000000000000000 and
;; #b111111111111111 and returns the bit value at that address location.

(rc address)

;; wc takes an address value between #b000000000000000 and
;; #b111111111111111 and a data value between #b0000000000000000 and
;; #b1111111111111111 and writes the data value to that address location.

(wc address data)
```

Reading and write to different address locations perform different functions on the board. The following is a list of the function of each of these addresses.

```
;; load0-15 loads instruction bits 0 through 15 into the microcode loader.

(define (load0-15 instruction-bits)
```

```
  (wc #b010100000000000 instruction-bits))

;; load16-31 loads instruction bits 16 through 31 into the microcode loader.

(define (load16-31 instruction-bits)
  (wc #b010000000000000 instruction-bits))

;; load32-47 loads instruction bits 32 through 47 into the microcode loader.

(define (load32-47 instruction-bits)
  (wc #b001100000000000 instruction-bits))

;; load48-63 loads instruction bits 48 through 63 into the microcode loader.

(define (load48-63 instruction-bits)
  (wc #b001000000000000 instruction-bits))

;; load64-79 loads instruction bits 64 through 79 into the microcode loader.

(define (load64-79 instruction-bits)
  (wc #b000100000000000 instruction-bits))

;; load80-92 loads instruction bits 80 through 92 into the microcode loader,
;; and downloads the instruction word in the microcode loader to the micromemory.
;; Instruction bits 93 through 95 are currently unused.

(define (load80-92 instruction-bits)
  (wc #b000000001111111 instruction-bits))

;; start-board starts the machine at the last loaded microinstruction, which
;; should start the machine at the beginning of the program in micromemory.

(define (start-board value)
  (wc #b011000000000000 value))

;; stop-clock! interrupts the board by stopping the system clock high.

(define (stop-clock!)
  (wc #b100000000000000 0))
```

```
;; wait is used to see if the machine is done processing and is in a waiting
;; state by reading the wait bit, thus determining if the host to access is
;; allowed to access memory.  If the value returned is odd, the machine is
;; not done processing.  If the value returned is even, then the machine is
;; in a waiting state and memory can be accessed by the host.

(define (wait)
  (rc #b100000000000000))

;; sram-high-offset gives the lowest data high-address location and
;; sram-low-offset gives the lowest data low-address location.

(define sram-high-offset #b101000000000000)
(define sram-low-offset #b110000000000000)

;; done tells the board that the host is done accessing memory, which allows
;; the machine to continue processing.

(define (done)
  (wc #b111000000000000 0))
```

Appendix F

Assembler Use

The assembler uses a register-transfer language that specifics for each instruction the "register" locations from which to read the operands for the given operation and the locations to which to write the result of the operation. There are three possible locations for data to be read from or written to: the data memory, the register file, and the feed-through latch. These are represented in assembly language as

```
;; data memory location saddress, where saddress is between 0 and
;; 4095, inclusive.  The two formats are equivalent.

(s saddress)
(sram saddress)

;; register file location raddress, where raddress is between 0
;; and 63, inclusive.  The three formats are equivalent.

(r raddress)
(reg raddress)
(register raddress)

;; feed-through latch

(latch)
```

Single and double precision operations are possible, but in double precision, addresses must be even because the assembler uses even number boundaries. In terms of instruction cycles, one single precision operation requires one instruction cycle, but one double precision instruction is converted by the assembler into the equivalent set of two single precision instructions, thus requiring two instruction cycles. We will deal mostly with double precision operations here.

There are four basic instruction formats allowed:

```
;; No operation.  Requires one instruction cycle.

(nop)

;; Loads register file location raddr with data in data memory location
;; saddr.  da instructions always require two instruction cycles.

(da (r raddr) (s saddr))

;; This is the main instruction.  Operation oper is performed on the
;; operands located in register file locations raddr1 and raddr2 and
;; assigns the result to register file location raddr.  The first
;; operand is the A-port operand and the second operand is the B-port
;; operand.  The locations for data operands and result destination
;; can be the same location or different ones.  The instruction stream then
;; proceeds to the instruction marked by :label.  There are a number of
;; optional fields and variations of the listing shown here.

(da (r raddr) (oper (r raddr1) (r raddr2)) (goto :label))

;; Tells the host machine that SMM is in a wait loop that promises not
;; to access data memory so that the host can access it.  Note that this
;; command is an assembly language instruction and is different from the
;; interface instruction wait.

(wait)
```

Although loading the register file from data memory and the execution of other operations can actually be performed together, the current assembler does not have that functionality. There has been some initial work done to provide the functionality, but it has not yet been completed due to the time constraints on this project.

The main instruction used has a number of variations and options that can be combined. Here are a number of the possible combinations:

```
;; Assigns result to multiple destinations.  Since the goto field has
;; been omitted, the program would proceed to the next instruction.

(da ((s saddr) (r raddr) (latch)) (oper (r raddr1) (r raddr2)))

;; First operand can be from the latch instead of register file.  The
;; condition bits of the result are latched by using lcc.

(da ((s saddr) (r raddr)) (oper (latch) (r raddr2)) (lcc))

;; Operation with one operand.  The condition bit cc1 is tested, and
;; if it is equal to one, a branch occurs to the instruction marked by
;; :foo.

(da (s saddr) (oper (latch)) (if cc1 :foo))

;; Condition bits of result are latched and the program goes to the
;; instruction marked by :do.

(da (r raddr) (oper (r raddr1) (r raddr2)) (lcc goto :do))

;; Equivalent to nop but allows use of branching field.  The location
;; none serves as a place holder for the assembler.  No actual
;; location is accessed.

(da (none) (none) (goto :loop))
```

Except for the nop instruction which is a single precision operation, all of the instructions that have been listed so far are double precision instructions, as indicated by the "da" at the beginning of the instruction. For single precision instructions, "a" must be used. A double precision instruction is expanded into two single precision instructions and hence takes as long to execute as two single precision instructions (two instruction cycles).

The rules for programming with the assembler are listed below. A sample program is given in Appendix D as an example of the rules of programming the Standard Map Machine.

- **Use of labeling**: An instruction is given a label so that it can be referred to in a the program as in a goto or if clause. The label must have a colon as the first character of the label name.

- **Use of multiplier/alu**: The load mode for the multiplier and the alu must be loaded at the beginning of the program. It may be changed later to change the rounding mode of the chips. Results of multiply operations are sent ten instruction cycles after the execution of the instruction to the appropriate destination to be written. Thus the new result will not be available for use until twelve instruction cycles later. For the alu, the result is sent twelve instruction cycles after the execution of the instruction so that the new result is not available until fourteen instruction cycles later. Using the latch as the destination is a special case, as mentioned below. Concerning instruction spacing, at least two instruction cycles are required between double precision multiplier operations, and if two multiplier operations are within twelve instruction cycles of one another, they must be separated by either two, six, or ten instruction cycles due to the pipelining requirements of the Weitek chips. Other separations will cause one of the operations to fail. Double precision alu operations can be run without separation in between them, but if there is separation between two alu operations and they are within fourteen cycles of one another, the separation must be an even number of instruction cycles. When interleaving the alu and multiplier operations, the only additional constraint is that a multiplier operation must have at least two instruction cycles between it and the

previous alu operation. The currently available multiply operations are * and −*. The currently available alu operations are +, -, cmp, f−>i, i−>f, and f−>is. The * operation does straightforward multiplication and the −* operation multiplies two numbers and negates the result. The + operation adds two numbers, the - operation subtracts the second operand from the first, the cmp operation compares two numbers (see instructions for branching), the f−>i operation converts a floating point number to an integer while i−>f does the opposite, and f−>is generates the page write address (see instructions for writing data to data memory).

- **Branching on conditions**: lcc latches the condition code bits coming out of the Weitek chips resulting from the operation performed. For a double precision instruction, the latching occurs on the first single precision instruction that it is expanded into. The condition codes can be tested using the if clause twelve cycles after the lcc has been performed (i.e. there must be the equivalent of at least twelve nops between the lcc and the if clause). The if clause tests the specified condition code bit and goes to the specified instruction if the condition is true. (Since the if clause is placed in the second single precision instruction resulting from the expansion of a double precision instruction and the lcc clause is placed in the first single precision instruction that results from the expansion of a double precision instruction, the spacing between double precision instructions can be just ten nops rather than twelve.) Thus far, only the alu comparison instruction has been provided by the assembler, although it is simple to add functionality. The comparison function cmp takes two operands as was shown above. To test if the first operand is less than the second, latch the condition code bits and test if cc0 is true. If the test is true, then the first operand is less than the second. To test if the first operand is greater than the second, latch the condition code bits and test if cc1 is true. If the test is true, then the first operand is greated than the second.

- **Use of feed-through latch**: The feed-through latch is used for fast serial computations by allowing the result coming out of the multiplier/alu to be directly fed back into it as the A-port input. The latch is used in the same way as a register file location, except that precise

timing is required. The result of a multiplier operation must be used exactly ten cycles later and the result of an alu operation must be used exactly twelve cycles later. Otherwise, the instruction will fail and the assembler will give a warning. Note that if there is a branching instruction in between the instruction that uses the latch as a destination and the instruction that uses the results from that latch as an operand, the assembler may still give a warning even though the instructions are separated by the right number of cycles.

- **Reading and writing data to data memory**: Data is read from data memory from the address location specified in the instruction. The range of addresses is even numbers 0 through 4094 for double precision data values. Data cannot be read ten cycles after a multiplier operation that writes the result to memory or twelve cycles after an alu operation that writes the result to memory. (This is also true for register file locations.) Data is written to data memory from the address location specified in the instruction and the page address calculated in the last f– >is operation. Page addresses range from 0 to 255 while write address locations for each page range from 0 to 14, even numbers only for double precision data values.

Instructions for assembling microcode, and downloading microcode and data to SMM have been implemented for use with the assembler. They are given below.

```
;; chasm-file assembles the list found in ``file.scm'' into the
;; instruction bits *chasm-bits*.  Note that ``file.scm'' must be in
;; list format.

(chasm-file ``file'')

;; download-code downloads the instruction bits in *chasm-bits* to SMM.

(download-code *chasm-bits*)

;; download-data downloads data-i in 64-bit IEEE floating point
```

```
;; format to data memory location-i.  Note that locations must be even
;; numbered ranging between 0 and 4094.  If no location is specified,
;; data is written into the next address location.  The initial
;; default location is address 0.

(download-data '((addr1 data1) (addr2 data2) ... ))
```

Bibliography

[1] Abelson, H. and Sussman, G. J. with Sussman, J., *Structure and Interpretation of Computer Programs.* M.I.T. Press, MA, 1984.

[2] Applegate, J., et. al., *A Digital Orrery.* IEEE Transactions on Computers, Vol. c-34, No. 9, September 1985.

[3] Berge, P., Pomeau, Y., Vidal, C., *Order within Chaos.* Hermann, Paris, France, 1984.

[4] Clinger, W. and Rees, J., ed., *The Revised^3 Report on the Algorithmic Language Scheme.* AI Memo 848a, Massachusetts Institute of Technology Artificial Intelligence Laboratory, MA, September 1986.

[5] *HP Series 200 Accessory Development Guide.* Hewlett-Packard Co., Colorado, 1983.

[6] Jensen, R. V., *Classical Chaos.* American Scientist, Volume 75, March-April 1987.

[7] Parker, T. S. and Chua, L. O., *Practical Numerical Algorithms for Chaotic Systems.*

[8] Higgs, M., *Advanced Schottky Load Management.* Texas Instruments Inc., 1987.

[9] Roylance, G. L., *Expressing Mathematical Subroutines Constructively.* AI Memo 999, Massachusetts Institute of Technology Artificial Intelligence Laboratory, MA, November 1987.

[10] Sussman, G. J. and Wisdom, J., *Numerical Evidence that the Motion of Pluto is Chaotic.* AI Memo 1039, Massachusetts Institute of Technology Artificial Intelligence Laboratory, MA, April 27, 1988.

[11] Yip, K. M-k., *Extracting Qualitative Dynamics from Numerical Experiments.* AI Memo 950, Massachusetts Institute of Technology Artificial Intelligence Laboratory, MA, March 1987.